SweetAda

Ada development environment based on

ARCHAEA PROTOCODE

SweetAda version 0.1 user's manual
(preliminar/incomplete)

Table of Contents

Introduction

Welcome to SweetAda.

SweetAda is an Ada-based development environment suitable for build lightweight applications in CPU-driven electronic devices. Developed from scratch with simplicity in mind, SweetAda tries to fill the gap between a simple, yet reliable, software/firmware control system and big, complex, OS-based multitasked environment.

In technical terms, SweetAda is a multi-platform self-consistent Ada ZFP-runtime-based executive without Annex-J support, without task/protected objects support. By self-consistent we means that the executive does not need low-level software layers.

Overall structure

The SweetAda system is composed of a main directory and some subdirectories which hold various software components. They are:
- application: generic user code
- clibrary: small-size C library
- core: machine-independent core software
- cpus: machine-dependent CPU low-level library
- drivers: I/O and peripherals components
- freeze: test tool to compare two builds
- libutils: utilities and scripts
- modules: various high-level libraries
- obj: build system output files
- platforms: platform-dependent low-level library
- rts: Ada run-time

SweetAda toolchains

The SweetAda system is distributed with GNU toolchains for every CPUs it can handle. Altough toolchains are quite generic, SweetAda does not employ neither the standard GNU FSF RTS source package nor the libgcc library.

The SweetAda RTS is a ZFP run-time that avoid to include subprograms from the original implementation, which is quite big and requires an underneath operating system.

The libgcc library, which is present like in every other standard toolchain, is bypassed and reimplemented in Ada. Only some low-level machine-language files are required, which are distributed in a separate package. This way, you have a system which is completely in control, and every bit of information is known. Code output from the build system is thus coming directly from the compiler, without extra source files inclusion. Anyway, the standard libgcc could be selected with a configuration variable.

Apart from assembler, compiler and linker, toolchains have the GNATMAKE utility and a “GCC wrapper” executable, which is an auxiliary tool used to manage object output files. The GCC wrapper is a small piece of code between the compiler driver and the compiler executable. It analyzes command line parameters, selects verbosity level and instructs the compiler executable to generate additional output files, like assembler listing and Ada expanded source code. There is a companion wrapper, the “GNAT wrapper”, but its use is limited to print brief output informations when GNATMAKE processes with’ed units.

elftool is a small utility that dumps the output object code. It is used mainly for a clean visual output in order to easily parse ELF sections.

The “core” complex

In Ada terminology, you can think of the core complex as an additional runtime system that sits on top of the ZFP runtime and expands the capabilities of the latter. The core complex has various packages, e.g.:

Bits	– low-level bits&bytes definitions and subprograms
Console	– generic input/output textual routines
MMIO	– generic low-level memory mapped routines
Memory_Functions	– low-level C-compatible routines

C-library

SweetAda has a very minimalistic C library. This C library takes no active part in SweetAda. Contrarily to all other environments, where Ada code runs on top of another piece of software (almost invariably a C-coded OS), in SweetAda even various low-level functions normally belonging to a standard C library are written in Ada. This small library exists for reference purposes, and as an aid in porting foreign C user code. Note that only very basic functions in ctype/stdlib/stdio/string hierarchies are implemented.

Optimizations

SweetAda provides a mechanism to allow user-defined implementations. Standard subprograms available in the core complex (but not limited to) could be bypassed by reimplement them.

In a similar way, some package and subprograms can be selected when speed is a priority. The package MMIO, as an example, exposes a complete range of procedures and functions to allow low-level I/O, like reading an Unsigned_32 value at a specific address. These subprograms are written in Ada and are not optimized. Furthermore, they cannot be inlined and thus can be used as access subprograms to be designated in drivers when you are configuring the methods to deal with I/O. But some CPU have a subset of MMIO that overrides the standard one. By rewriting the subprograms with inline assembler instructions, speed increase could be easily obtained. Beyond that, some CPU have a second package CPU.MMIO with these inline assembler fragments already written. With’ing CPU.MMIO instead of MMIO lets the user directly use them.

Configuration

The SweetAda build machinery is Makefile-based (but you can use an alternate GPR-style build model -- this is enabled by a configuration variable, see below). Every configuration file is a Makefile fragment that contains assignments in that syntax. The various configuration files are hierarchical and are loaded by the master Makefile in the appropriate sequence.

The master Makefile try to detect the type of machine it is running on, and configuration of SweetAda begins by reading the top-level directory configuration file “configuration.in”. Essential variables are set to a default value, like the path to the toolchain prefix, the build model, RTS type, profile, and so on.

One important aspect is that configuration variables could be introduced (and could override pre-defined ones) at various stages of the configuration. Obviously, further and more specific configuration files are loaded in sequence from the very few basic information found in the top-level configuration.in.

Then, if it exists, a “kernel.cfg” file from the top-level directory is loaded. kernel.cfg contains the hardware platform which will be the build target. If kernel.cfg is missing, you can specify the platform (and eventually the subplatform) by assigning the shell environment variables PLATFORM and SUBPLATFORM, which in that case take precedence over an existing file. PLATFORM should be a valid directory in the “platforms” subdirectory, and SUBPLATFORM should be a valid subdirectory of the PLATFORM directory, named “platform-<SUBPLATFORM>”.

Once you choose a platform, you can create the kernel.cfg with the Makefile auxiliary target “createkernelcfg”.

```
$ PLATFORM=PC-x86 SUBPLATFORM=QEMU-ROM make createkernelcfg
```

(NOTE: for a cmd.exe shell you have to use a “SET” command).

Following this command, you can check the presence of a correct kernel.cfg file. Once kernel.cfg does exist, it will be used until a “make all-clean” command, which erases every configuration file.

Issuing a new configuration erases the old one by executing an implicit “all-clean” Makefile target.

After the top-level configuration file is loaded, and thus Makefile knows the platform, the configuration.in found in that directory is loaded (e.g., ./platforms/ML605/configuration.in). Here you can specify nearly everything about the target machine, like the CPU model, fine-tuning compiler switches, run- or debug-related auxiliary scripts, and so on. If you need to create some information that will be used by external scripts, this is right place. Pay attention to export every variable you are creating not known by the build system, otherwise the external scripts cannot access it.

Having knowledge of the hardware configuration, the configuration.in in the appropriate CPU hierarchy is loaded (e.g., cpus/MIPS/configuration.in).

Once the CPU is well defined, the right files can be taken into account. The standard toolchain is selected by default, but you can pick another one by, e.g., overriding the variable TOOLCHAIN_NAME_<CPU> in the configuration.in of the platform directory.

Then, the top-level “Makefile.tc.in” file is taken into account. This a very important part of the global configuration scenario, since it is responsible of the setup parameters for the compiler toolchain, compilation switches, warning switches and executable paths. If you want to suppress checks, warnings, or change programming style layout, you can comment/uncomment the appropriate compiler switch.

Now that the configuration setup is complete, the build system is ready for the next step, the configuration target proper.

```
$ make configure
```

The “configure” Makefile target will call the homonym target in every sub-Makefile concerned, e.g., “core”, “modules”, “drivers”, and the platform. This way, fixing of files that should be included in the build or a pre-processing from a template file can take place. As an example, files from SUBPLATFORM specific directory could be symlinked in the platform set of files, the “gnat.adc” file will be created from the template file “gnat.adc.in”, and so on.

The build system will reply by writing a brief configuration status:

```
Configuration parameters:
PLATFORM:          PC-x86
SUBPLATFORM:       QEMU-ROM
CPU:               x86
CPU MODEL:         i586
SWEETADA_PATH:     /root/project/sweetada
TOOLCHAIN PREFIX: /opt/sweetada
TOOLCHAIN NAME:    i686-sweetada-elf
GCC VERSION:       9.3.0
GCC SWITCHES:       -march=i586 -Wa,-march=i586
GCC MULTIDIR:      .
RTS ROOT PATH:     /root/project/sweetada/rts/i686-sweetada-elf
RTS PATH:          /root/project/sweetada/rts/i686-sweetada-elf/.
LD SWITCHES:
OBJCOPY SWITCHES:  --output-target=binary --gap-fill=0x00
OBJDUMP SWITCHES:
RTS:               ZFP
PROFILE:           ZFP
BUILD MODE:        MAKEFILE
```

You can now build the system:

```
$ make all
```

The final product of the build will be the file “kernel.o”, which contains an ELF image of the system.

Once the final ELF object is created, you can execute the “postbuild” Makefile target, which extract code/data sections from kernel.o, and produces a binary file suitable for ROM programming, JTAG download, and so on. This target uses the “objcopy” Binutils utility, so appropriate configuration switches could be selected by means of the OBJCOPY_SWITCHES variable in the platform’s configuration.in file.

Note that the postbuild target is a two-phase process. The first phase creates a binary files full of informations, then the postbuild target specified in the platform’s Makefile is executed. This way you can customize the final binary file or

take other actions, like add custom data, place the output file in a server's TFTP directory, and so on.

Some platforms are virtual, i.e., they use an emulator, and in some cases there is no need to execute the postbuild target, because the emulator could use directly the kernel.o file.

The SweetAda Makefile build system can be queried. The special Makefile target PROBEVARIABLE outputs the value of a configuration variable, so, e.g., you can execute a command line like:

```
$ make VERBOSE= PROBEVARIABLE=PROFILE make -s probevariable
```

and the build system will print the value associated with the variable PROFILE. A script could intercept the output by means of a redirection and use this information. Note that the VERBOSE variable is set empty and the the "-s" option is passed down to the Makefile so the output is not cluttered with other non-interesting output.

Configuration files

Configuration of SweetAda is done with two configuration files: "main" (or "system") configuration file, and a "platform" configuration file. The "main" configuration file defines general, target-agnostic characteristics of the target software, like stating Ada95 or Ada2012 mode, choose the optimization level, and so on; the "platform" configuration file defines in greater detail how the target software is being generated, and various parameters affecting low-level behaviour. Both files are, as a matter of fact, Makefile fragments that are going to be incorporated in the build system at compile-time. Being the SweetAda build system based on GNU Make, many facilitations are possible (like variable substitutions) and, if correctly arranged, configuration items can be made modular and easily changeable.

The final output product of SweetAda compilation is the file "kernel.o". Its format depends on the type of toolchain used, but it is almost exclusively an ELF-class object file. As part of the link process, the master Makefile invokes the target "postprocess" (in the platform's Makefile) in order to create a binary file suitable for real code execution. Apart from some virtual platforms that use directly the ELF executable, this file is a binary image used for creating a Flash EPROM or the like.

Generally speaking, Eclipse does not recognize shell scripts as valid executable files, so "startkernel" is actually a wrapper executable that gives Eclipse an idea about which program is going to be run (incidentally, when a virtual platform is targeted, that program is really a native executable from the host's point of view – it is an application that mimics the functionalities of a machine for which some code has just been written).

IOEMU scripting

The IOEMU facility allows configuration of a target emulator machine.

Strings
String exists when you write literals inside double-quotes. Use a backslash to enter a special characters.

Variables
Variables are strings. Numeric values are converted to string. If you want to dereference a variable, use a "$" prefix followed by the name of the variable. Use braces to isolate variables when concatenating them with literals, like in "${PORT1}". Note that dereferencing takes place only within double-quotes.
There are three types of variable in IOEMU scripting: environment, global and local. Environment variables are available all the times, and are automatically imported. You can create an environment variable (being passed to child executable) with the ENV <variable> <value> directive. Global variables are created with the SET <variable> <value> directive. Variables declared within sections are local to that scope and ceased to exist when the parser leave the section.

Boolean expressions
NOT AND OR == DEFINED()

IOEMU creates some implicit variables during its execution.

__THISFILE__ carries the name of the configuration file being processed. So you can export this variable and pass it to a emulator, that thus could use this same configuration file. Note that this is quite mandatory, because SweetAda emulators loads the IOEMU shared library and the configuration file using environment variables.

__SERIALPORTDEVICE__ (Linux only) carries the name of the virtual device associated with an emulator physical serial port. When you are using QEMU there is no need to use this variable, because the emulator maps physical serial

ports to known TCP ports declared with command line arguments, but with the FS-UAE emulator the IOEMU layer creates on-the-fly a virtual serial port that cannot be known before the execution.

SECTION
Introduces a section. Section names could be “IOEMU” (to configure the IOEMU layer) or an I/O port or communication peripheral exported by the target machine being emulated.

ENDSECT
Declares the end of a section.

IF-THEN-ELSE-ELIF-ENDIF
Classic decision structure. The test condition must be a boolean expression. IFs structures can be nested. Note that IFs structures can be used only inside sections.

ARGS
ARGS is a special directive. When you declare an ARGS directive, strings are added sequentially to the array that carries arguments for the next EXEC/EXEA command. So you can add arguments to the desired executable at various points of a section, and decided which arguments must be selected. Once the EXEC/EXEA instruction is processed, the ARGS array is reset to empty.

EXEC
Executes an executable. The arguments are those accumulated by means of previously processed ARGS commands. The script control flow does not proceed further until the executable returns.

EXEA
Same as EXEC, but the executable is launched in the background, so script control flow can proceed almost immediately.

SLEEP
Suspends the control flow execution for an amount of time.

FAQ

Q:
How about the SweetAda logo?
A:
The SweetAda logo (better: the “badge” -- because it isn’t a legally registered item) is a sketch of venerable Pioneer 10 space probe. Built in early 70s by TRW Inc. in Redondo Beach, CA, it is the first human-made artifact which left the Solar System. Pioneer 10 was launched on March 3, 1972 from Space Launch Complex 36A in Florida, and its mission lasted until January 23, 2003, when communications were lost because of the loss of electric power for its radio transmitter, with the probe at a distance of 12 billion kilometers (80 AU) from Earth. Chances are it is almost functional yet, travelling into deep interstellar space, alone. A magnificent model of technological elegance and superior reliability which deserves to be imitated.

Q:
Which is the difference between SweetAda and Archaea? Do they refer to the same thing?
A:
SweetAda is the whole build system, including toolchains, RTSes, host machine utilities, debugger facilities and so on. Archaea is, more or less, the Ada support code which forms the main part of the final executable: the CPU/platform libraries, the core complex, and the drivers. As a shorcut, when no confusion could arise, SweetAda and Archaea are synonyms.

Q:
Why the rusweetada launcher?
A:
To begin with, to make things simple. Most of the time you have to write shell scripts and complicated wrappers around executables to configure the environment. runsweetada try to centralize these annoying activities with a uniform mechanism. Besides, it could be difficult for some third-party environment (say, e.g., Eclipse) to recognize a shell script as an executable to launch. In Windows, for example, you cannot launch a Bash script directly, because it isn’t a system registered binary. As a final note, runsweetada works like a front-end configuration for the IOEMU environment.

Q:
It is not clear why so much emphasis about not using GCC’s libgcc.
A:
GCC uses libgcc as a target-specific library for low-level operations, and when it is not efficient to emit inline code for (e.g., very long integer multiply). libgcc also contains various OS-specific routines that enlarge its memory footprint, resulting it in a rather sizeable piece of software that your system, as matter of fact, has no knowlegde of. This is not acceptable, because every bit of code must be fully traceable, and Ada ZFP runtimes should not have references to external function libraries. Only few strict CPU-dependent functions are integrated in SweetAda (but always at the source code level!), making it a complete and well-defined source-code-controlled ensemble. (Note: this not even applies for all CPUs: for example, x86 CPUs do not need libgcc routines at all, because GCC almost exploits the CISC instruction set in “native” mode.)

Q:

SweetAda claims it can be burned in a Flash EPROM and placed in a PC motherboard. It’s obviously fake news.
A:
A PC motherboard is just another embedded device that happens to have a keyboard, a mouse and a video monitor; it is only a consumer product that you can find in the store down the street (just for the records, the difference between a common PC and a PC104-style embedded board is basically the PCB format). Once SDRAM and PCI basic devices are initialized, it can be used like any other development board. Naturally, only a few old i586/PIIX PCI motherboards are supported, because support for leading-edge multi-core CPUs and modern PCI bridges is non-trivial and time-consuming. This was made for 1) testing SweetAda on more machine typologies as possible, and 2) because it’s relatively easy to develop on PC-class machines. See PC-x86 subplatforms for reference.

Q:
It is not clear what a user should do in porting SweetAda to a new platform.
A:
SweetAda cannot support every physical platform/device out of the box. The bare minimum that a user should do in order to let SweetAda run is deploying a functional memory subsystem, so the kernel can use RAM memory for its internal workings (e.g., stack and static data). This could be as simple as doing nothing if RAM is made out of static devices (so you just have to trivially specify correct address spaces in a linker script). In the case of DRAM/SDRAM devices, RAM memory initialization should be carried out in the startup-memory.S source file – then the kernel will be able to perform procedure and function calls. If you are developing a proprietary embedded device, consult your hardware engineer/consultant for details.

Q:
The build system is rather convoluted.
A:
Mainstream use of GNAT compiler and tools, when dealing with build process, does generally mean the use of GPRbuild. Makefiles are much more understandable throughout software community, and GNU Make is a tool which is independent from the toolchain used, at least to a better grade when also considering GPRbuild.

Generic notes

A tribute to Augusta Ada King-Noel, Countess of Lovelace (née Byron; 10 December 1815 - 27 November 1852), believed to be the first computer programmer. Centuries ahead of her time, she dealt with the Analytical Engine, a machine being developed by English Mathematician Charles Babbage. Her picture in the first page of this manual comes from the original file once stored at the Ada Joint Program Office site.

A tribute to Jean David Ichbiah (25 March 1940 – 26 January 2007), initial Ada language chief designer.

This document use STIX fonts from http://www.stixfonts.org.

Acronyms

BSP	Board Support Package
CPU	Central Processing Unit
ELF	Executable and Linkable Format
FSF	Free Software Foundation
GCC	GNU Compiler Collection
GNAT	GNU New York Ada Translator
MMU	Memory Management Unit
ZFP	Zero-FootPrint