

CarbonKernel

Real-time Operating System Simulator

Version 1.4

User Manual

October 2001

Contents

1.Introduction	5
1.1Overview	5
1.1.1What is CarbonKernel?	5
1.1.2What is the tool for?	5
1.1.3What are the event-driven simulation approach's advantages?	5
2.Installation from a binary distribution	7
2.1User Environment Self-Test	7
3.Description of the simulation system	8
3.1Overview of the system architecture	8
3.2Hosting real-time applications in the simulation system	9
3.2.1Working with the code instrumenter	9
3.2.2Removing machine-level dependencies	12
3.2.3Startup code	12
4.Setting up a simulation	13
4.1Instrumenting a C/C++ application with ckcc	13
4.1.1Time Progression and Vectoring	13
4.1.2Instrumenter's Options	13
4.1.3Locally Controlling Preemption and Vectoring.	15
4.1.4Source Code Adaptation Requirements	16
4.1.5Preprocessor Signatures	17
4.2Ongoing Example: IBC Example	17
4.2.1Introduction	17
4.2.2Distribution	18
4.2.3Generating the Executable	18
5.Configuring the Simulation	19
5.1The Librarian	19
5.1.1Simulated configurations	20
5.1.2Typical Architecture	21
5.1.3Loading Simulation Modules	21
5.1.4Creating/Copying a Model Instance	22
5.2Modifying/Renaming a Model Instance	22
5.3Destroying a Model Instance	24
5.4Defining a Simulated Configuration	24
5.4.1Associating Nodes	24
5.4.2Linking Add-ins to the Simulation	25
6.Simulation Project	27
6.1Creating a Project	27
6.2Modifying a Project	28
6.3Selecting a Project	28
7.Control Panels	29
7.1Message ports	29
7.2Dataports	29
7.3Magnets	29

7.4 Creating Control Panels 29

8. Editing the project settings 33

- 8.1 Configuring Event Sources 33
 - 8.1.1 Events, Interrupts and the Simulation Scenario 33
 - 8.1.2 Event Source Settings Syntax 33
 - 8.1.3 Generating Events Automatically 33
 - 8.1.4 Generating Events Manually 34
- 8.2 General Simulation Settings 35
- 8.3 Simulation Tool Parameters 36
- 8.4 Operating Options 37

9. Exporting/Importing the work environment 39

- 9.1 Exporting/Importing the current project 39
- 9.2 Exporting/Importing the models library 39

10. Running a Simulation 40

- 10.1 Interactive Execution 40
 - 10.1.1 Simulation Monitor 40
 - 10.1.2 Using the built-in Debugger 46

11. Displaying Statistics Graphs 56

- 11.1 Selecting Graphs 56
- 11.2 Display by Type of Graph 57
 - 11.2.1 Time Curves 57
 - 11.2.2 Composite Curves 57
 - 11.2.3 Histograms 57
- 11.3 Controlling the Simulation 57
 - 11.3.1 Continuing/Stopping Simulation 57
 - 11.3.2 Setting Breakpoints 57
 - 11.3.3 Displaying and Checking Breakpoints 58
- 11.4 Scale Compression 59
 - 11.4.1 Y Compression 59
 - 11.4.2 X Compression 59
 - 11.4.3 Zoom In 60
 - 11.4.4 Zoom Out 60
- 11.5 Composite Curves 60
- 11.6 Placing Graphs 60
- 11.7 Selection and Cross Hairs 60
 - 11.7.1 Actions on Graph Sections 60
 - 11.7.2 Using the Cross Hairs 61
- 11.8 Other Local Functions 61
 - 11.8.1 Local Time Curve Functions 62
 - 11.8.2 Local Histogram Functions 62
 - 11.8.3 Common Functions 63
- 11.9 General Options 64
 - 11.9.1 Adjusting Abscissas 64
 - 11.9.2 Scroll Lock 64
 - 11.9.3 Auto-Select Color 64
 - 11.9.4 Auto-Save Session 64

12. Using the Terminal Console 65

- 12.1 Recording an Interactive Session 65

12.2	Replaying an Interactive Session	66
12.3	Manually Creating a Replay File	66
12.3.1	Format for Time Based Replay ("normal" mode)	66
12.3.2	Format for Immediate Replay ("raw" mode)	67
12.4	Saving Terminal Outputs	67
12.5	Deleting the Terminal During Simulation	67
13.	Command Lines	68
13.1	ISE Start Options	68
13.2	Simulator Start Options	69

1. Introduction

1.1 Overview

1.1.1 What is CarbonKernel?

CarbonKernel is a real-time operating system simulation tool based on event-driven simulation techniques. The main idea is to provide a common framework for simulating the behaviour - from the application's standpoint - of one or more RTOS flavours.

CarbonKernel features a virtual RTOS including a comprehensive set of generic services, allowing to build specific RTOS flavours on top of it. The result of such specialization is called an RTOS *personality*. Each simulated RTOS is expected to display the same kernel API as its embedded counterpart the applications can depend on. The eCos kernel for which a simulation model is available will be used to illustrate the CarbonKernel features later in this document.

The event-driven general-purpose simulator underlying CarbonKernel's virtual RTOS is **FROGS**. It is available as a standalone package.

1.1.2 What is the tool for?

CarbonKernel is a versatile simulation system for running embedded real-time software on a workstation, thus making the whole process of writing and testing your code much faster and easier when compared to working in the target environment.

Using CarbonKernel, embedded applications that use the standard services displayed by the kernel API of a RTOS can run in a host environment. The basic idea behind the simulation approach is to provide a comfortable development and testing environment on a workstation for distributed and/or complex embedded applications. CarbonKernel can simulate a global application code running simultaneously on several RTOS instances with different characteristics within a single session.

The compilation toolchain used to build the simulated application is the workstation's native one. Using CarbonKernel first, you don't need to depend on a cross-compilation toolchain for implementing and testing the target-independent code. Due to the fact that CarbonKernel works at source code level, it is not tied to any specific target processor or hardware.

1.1.3 What are the event-driven simulation approach's advantages?

An event-driven simulator does not rely on the hosting machine's system clock but instead provides a simulated timeline to schedule the events that occur when the application code is running. This means that a given application scenario can be repeated very simply an infinite number of times, with no perturbation from the outside world (e.g. current load average of the simulating host and so on).

Not depending on the host's idea of time also means that portions of code can be executed at no time charge, such as instrumental code.

Because it can trigger user events automatically or manually at selected times when the application is running, CarbonKernel is very good at creating test and

stress situations for the application otherwise nearly impossible to reach (at least willingly...) on a real target.

To track these situations, **CarbonKernel** offers an integrated simulation monitor and symbolic debugger. These tools can help you to isolate the different application contexts by focusing on specific execution pathes (thread scope, target scope or overall multi-target scope) and show and/or modify the status of the simulation objects involved.

CarbonKernel *is not a processor instruction set simulator*, but its is a RTOS simulator. This means that **CarbonKernel** is not suitable to obtain accurate performance footprints for the simulated application. On the other hand, not only can it simulate different RTOSs' personalities, but it can do the same for their entire environment via **FROGS** native simulation models (i.e. Add-ins). For example, communication protocol models can be attached to the simulator to improve simulation accuracy and so use realistic settings to evaluate the application's behaviour.

2. Installation from a binary distribution

CarbonKernel binary distributions are usually available in `tar.gz` or `tar.bz2` format. Just inflate the selected archive appropriately into your **CarbonKernel** installation directory, preserving file permission bits (i.e. One should use **tar's** `p` flag).

A source distribution creates the same file hierarchy when installing the built executables and libraries. The `--prefix` option passed to the top-level `build-all.sh` script should point to the installation directory. By default, `/usr/local/ck` is used as the installation prefix.

Each **CarbonKernel** user's environment variables `PATH` and `LD_LIBRARY_PATH` should be updated to include the access paths for the distribution's `bin` and `lib` directories, respectively.

2.1 User Environment Self-Test

The standard distribution contains a program called `selftest`, which can be used to check that the current environment is suitable for using **CarbonKernel's** simulation tools. This executable can be found in the `bin` directory.

Under the user account being tested, run the following command:

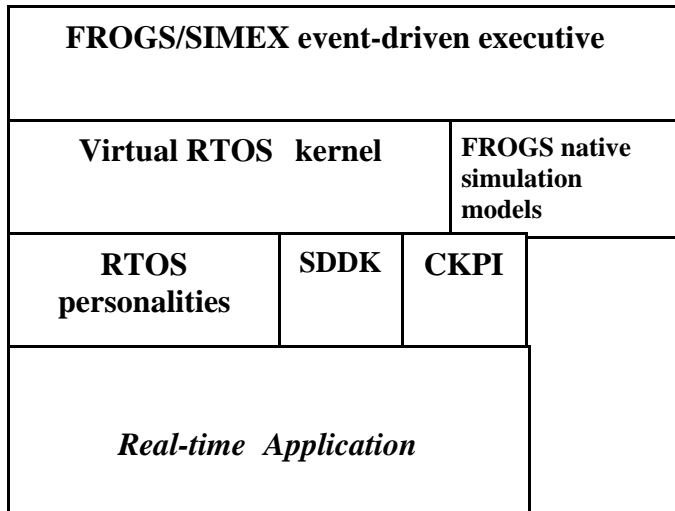
```
$ selftest
```

After the command has been run, an analysis shows whether the process was successful or not. If an error occurs, information for resolving it is automatically displayed relative to the seriousness of the problem.

3. Description of the simulation system

3.1 Overview of the system architecture

The CarbonKernel's system architecture is layered as follows:



- SIMEX is a versatile, object-oriented simulation executive featured by the FROGS simulation framework, implementing an event-driven scheduling system. Time-based activities are executed on behalf of pseudo-concurrent threads, which can exchange messages and synchronize themselves using a pre-defined set of extensible simulation objects. The global simulation clock is managed at this level.
- The virtual RTOS kernel (aka VRTOS) implements an extensible real-time kernel on top of the SIMEX executive. It is an event-driven simulation model that can be connected to other native simulation models. The virtual RTOS kernel provides a comprehensive framework to implement personalities of embeddable real-time operating systems, aimed at simulating their original kernel APIs. Its architecture allows multiple virtual RTOS instances called *nodes* to run concurrently in a single simulation process. The concept of a node should be understood as the CarbonKernel's representation of a target hardware hosting an instance of a RTOS personality. Whilst all these targets are synchronized on the unique SIMEX executive clock, they may locally define the duration of a single clock tick of their own real-time clock.
- FROGS native simulation models can be added to the simulation system to implement the behaviour of virtually any kind of activity, whether hardware or software. These models are designed to act as their real world counterparts do. Depending on the expected simulation accuracy, it's the model designer's job to define what are the real world item's characteristics that really need to be simulated. The virtual RTOS kernel should be seen as a FROGS native model displaying the behaviour of common real-time system objects (e.g. Such as threads, synchronization objects, schedulers and so on). For instance, one could add hardware (e.g. UART), protocol software (e.g. HDLC) or even complex system (e.g. DBMS) component models with interfaces to access them from the application layer. Those models would interact with the overall simulation process by triggering events and running

threads of control in parallel to the RTOS simulation performed by the VRTOS.

- ▣ The SDDK interface (i.e. Simulated Driver Development Kit) is used to develop simulated device drivers. The main goal of a simulated device driver is to implement the simulation counter-part of a "real" driver accessing "real" hardware for the application, by providing a normalized way of sending and receiving data to/from a pseudo-device faking the real hardware during the simulation process.
- ▣ The CKPI interface (i.e. CarbonKernel Programming Interface) gives the application, the SDDK and the FROGS native models access to a comprehensive set of system services exported by the virtual RTOS kernel. Because they do not depend on any specific RTOS personality, these services can be used inside reusable simulation components.
- ▣ A RTOS personality is a simulation model built on top of the virtual RTOS kernel. It exports the programming interface of an embeddable RTOS.
- ▣ The real-time application is the embedded software system one may want CarbonKernel to host on a workstation. Such application may request system services of a given RTOS using its public kernel API, which should be simulated by the appropriate RTOS personality on top of the CarbonKernel's virtual RTOS. C and C++ written applications are currently supported by the instrumentation tool.

3.2 Hosting real-time applications in the simulation system

Real-time applications written in C and/or C++ languages can be hosted by the simulation system. The current discussion assumes that you intend to host an existing target-based real-time application into the CarbonKernel environment. However, if you are rather writing a new application from scratch using CarbonKernel which is designed to be ultimately embedded into a real target environment, some information below could be of great help too.

Three major issues must be addressed for the hosting process to succeed:

- ▣ A mean must be found to charge the application code for execution time, given that CarbonKernel is not a machine-code simulator, but rather a RTOS simulator (because we do want strictly reproducible, timely behaviour of the application across simulation runs, depending on the host computer's idea of time is definitely not the right thing to do either). CarbonKernel's code instrumenter (i.e. **ckcc** for C and C++ source applications) provides all the needed support to complete this task.
- ▣ It should be allowed for multiple instances of the application code to run concurrently in a single (process) address space. Once again, the code instrumenter prepares the application code to support this feature.
- ▣ Finally, the hosted part of the application must not contain machine-level dependencies, such as portions of code written in assembly language for the target processor, or assume any pre-defined memory mapping. Such code will just not assemble or run, and if it does, it would probably cause fatal runtime exceptions. Removing or masking such machine-level dependencies in the context of using the simulator is the user's responsibility.

3.2.1 Working with the code instrumenter

The code instrumenter prepares your application code to be run by the simulation kernel. The CarbonKernel's instrumenter you should use depends on the programming language of your code. As of now, **ckcc** can be used for C/C++ source applications. One should note that in no way the instrumenter changes the functional behaviour of the application. It only adds code to bind the application to the simulation system.

The code instrumenter should be interposed at compilation and link time just before the standard compiler and linker programs. You usually just need to set your *Makefile* variables in order to use **ckcc** instead of “**gcc**” to compile and link your application. By default, **ckcc** invokes **gcc** as the final compiler to produce the object file from the instrumented source. Please refer to the **Instrumenter's options** section to learn how this behaviour can be changed.

For instance, here are the commands needed to compile two C files, namely *foo.c* and *bar.c*, then link the obtained object files to produce a final simulation executable, using the eCos simulation model:

```
ckcc -g -c foo.c
ckcc -g -c bar.c
ckcc -o app foo.o bar.o --rtos=ecos
```

3.2.1.1 Updating the global simulation clock

The instrumenter adds a call to an internal routine before each executable statement found in the original source code. This routine is meant to signal a new simulation clock tick to the virtual RTOS kernel, each time a source statement has been executed, and the next is about to start. The simulation clock tick should not be confused with the real-time clock tick which occurs after each simulated RTC interrupt on a given node.

When the simulation clock tick is signaled, the SIMEX's global clock is advanced by a quantum of time, after all events whose scheduling times are prior to the new clock value have been processed. The duration of the simulation clock tick is determined on a per-node basis, by a special setting called the *Target Warp* factor, ranging from 0 to 10. In other words, changing this value directly affects the perceived processing speed of a simulated node, independently of the other nodes'. The higher the *Target Warp* factor, the shorter the time quantum charged per instruction. The time quantum is computed in microseconds from the *Target Warp* factor by the following formula: $1 / \exp(\text{factor})$. For instance, a node (i.e. Target board) configured with a *Warp* factor of 3.7 will be charged for $1 / \exp(3.7) = 0.02472$, that is to say 24.72 (simulated) nanoseconds for each source statement.

This is a fundamental point to catch in order to fully understand the way CarbonKernel deals with the simulated time: a fixed quantum of time defined on a per-node basis is charged for each source statement executed. This is obviously not suitable for obtaining accurate performance footprints, but this is enough to preserve the time sequence of real-time events as they occur. Raising or lowering a node's *Warp* factor combined to the automatic event triggers (e.g. Interrupts) can also help you stressing your application, such as testing for correct critical section enforcement and so on.

This time management scheme gives the multiple nodes in your simulation a single common time frame, independent from the host's idea of time. This makes your application's behaviour strictly repeatable between simulation runs, which is a very desirable feature in debugging complex systems. Moreover, you have absolute control

(except regression though) over the key component of an event-driven simulation system: the clock.

3.2.1.2 Allowing multi-node execution

The code instrumenter turns regular data with static storage attribute found in the original code into per-node variables, thus allowing the compiled form of a single source code to be traversed by multiple node contexts within a single simulation process, each of them having a private address space to store non-automatic, non-constant data. This way, you can instantiate a given node multiple times in a single simulation configuration to build a symmetrical runtime configuration.

The instrumented version of a given source file can be collected using the `--save-temps` option when invoking `ckcc` for compilation. As far as C and C++ source codes are concerned, each global variable or static local variable becomes a pointer to an array of pointers to the original data type, i.e. a kind of data vector. Each reference to such variable in the code is then tweaked to be indexed on the running node's identifier (which is a 0-based ordinal value), thus selecting the node's private address space appropriately. For instance, here are the changes applied to a global pointer declaration and to a statement referencing it:

```
/* Here is the original version... */

struct superblock {
    cyg_sem_t lockSema4;
} TheBlock = NULL;

netshared int SharedInt = 1;

void lockSuperBlock ()
{
    cyg_semaphore_wait(&TheBlock->lockSema4);
    SharedInt = 0;
    ...
}

/* Here is the instrumented version... */

struct superblock {
    cyg_sem_t lockSema4;
} (*(TheBlock));

int SharedInt = 1;

void lockSuperBlock ()
{
    cyg_semaphore_post(&(*(TheBlock[cknid__]))->lockSema4);
    SharedInt = 0;
    ...
}
```

It seems obvious reading this fragment of code that the `cknid__` integer variable contains the running node's identifier. Because you have no guarantee that the CarbonKernel's internal naming scheme will not evolve in the following revisions of the system, you definitely should call the `ckGetNid()` service to get this information rather than accessing this variable directly.

You may have noticed that the `SharedInt` integer variable was not changed by the instrumenter, thus keeping its value common to all node contexts. This is the effect of the `netshared` special data qualifier defined by the code instrumenter. Passing the `--netshare` option to `ckcc` when compiling the source file makes this qualifier implicitly applied to all data declarations found in the module. You may use it if your runtime configuration is limited to a single node, but there is absolutely no obligation to do so.

You may also have noticed that the initializer part of the data declaration (i.e. `... = NULL`) has been removed by the instrumenter. In fact, it has been memorized and used during the last phase of the instrumentation process when the data vectors construction code is emitted in a separate hidden routine. This initializer will be applied dynamically to each instance of the original data type pointed to by the cells of the data vector.

3.2.2 Removing machine-level dependencies

It is usually admitted that most part of an embedded application should be written using a high-level programming language such as C or C++, leaving the assembly language to the very low-level parts of the Board Support Package.

The Board Support Package usually provides machine-level hooks to bind the RTOS to the target hardware. In most cases, this support is not needed in a simulator like **CarbonKernel** which leaves the whole burden of managing the real low-level resources to the host operating system.

However, it may be useful or simply required for some hardware and/or software activities to be simulated, such as interrupts, device driver i/o, specific hardware widget behaviour and so on. Some may even find useful to simulate LAN and/or WAN activities between multiple nodes in a single simulation. There are answers for each of these concerns; some are pre-defined in the simulation kernel (e.g. interrupt management), some can be supported by specifically designed programming interfaces (e.g. SDDK-based device drivers), others are provided by **FROGS** native simulation models.

3.2.3 Startup code

3.2.3.1 Cold initialization code from BSP

The cold initialization code from the BSP should probably not be run by the simulator. It is usually aimed at attaching the embedded RTOS to its hardware environment.

3.2.3.2 Dynamic initialization of global data objects

As we discussed earlier, the code instrumenter turns all non-constant data declaration defined in the program's global scope into data vectors. As a part of its warm initialization procedure, the simulation kernel locates and initializes each vector appropriately with instances of the originally declared data types.

Because the C++ language allows code to be executed for building global class instances during the early initialization phase of the process through the use of static constructors, the simulation kernel ensures that a set of systems services callable on behalf of the initialization context can be issued to the RTOS' kernel API exported by the current node. The members of this set are defined by the specifications of the programming interface of the simulated RTOS personality.

The dynamic initialization of global data objects is performed before the simulation kernel calls the entry point of the first activated node.

3.2.3.3 Application boot code

The simulation kernel initiates each node activity by calling an entry point. This entry point is usually defined as a configurable parameter by the graphical configuration window of the node. For the CarbonKernel's eCos model, the user entry point is specified by the USER_START_ROUTINE parameter.

4. Setting up a simulation

The first stage in setting up a simulation consists of generating the executable program, called the *simulator*, which will include the following parts:

- ▣ The instrumented application code.
- ▣ The CarbonKernel's system libraries, including the VRTOS.
- ▣ One or more RTOS personality libraries exporting the appropriate kernel API(s) to the application.
- ▣ A set of optional FROGS native simulation models (i.e. Add-ins).

The first operation is carried out by the CarbonKernel source instrumenter for the language used to code the application (e.g. `ckcc` for C and C++). The other three elements are added to the application code by the linker, which is also driven by the instrumenter.

4.1 Instrumenting a C/C++ application with `ckcc`

The source instrumenter is a vital link in the CarbonKernel development toolchain. Its role consists of preparing the application code for running in the CarbonKernel simulator context and then acting as a relay to pass it to the traditional compilation process. However, instrumenting never changes the application's functional characteristics.

In other words, `ckcc` precedes the compiler in the source code preparation phase before object files are generated and when the linker is run. The traditional compilation process is still responsible for these two operations. Whenever application code is written in C or C++, the `ckcc` instrumenter must be used.

CarbonKernel C/C++ language support is provided by using a modified version of the GCC/EGCS compiler in order to use its instrumentation function. However this does not mean that a specific version of GCC/EGCS, or even GCC/EGCS itself, must be used to generate the application, as the instrumenter engine produces modified source code, and does not directly produce the final object file. However, `ckcc` manages the different stages needed to produce the object file from the original source file.

4.1.1 Time Progression and Vectoring

One of the main functions of instrumenting by `ckcc` is that each C/C++ source code statement is preempted; another important effect of instrumenting is that non-automatic data is vectored (vital when simulating several concurrent nodes). A separate instance of every item of non-automatic data (i.e. global variable or local static variable) in the application code must be created for each simulation node context, so that the same code can be traversed by multiple threads running in parallel on distinct nodes.

All vectoring of a file can be deactivated by using the `--net-shared` option in the instrumenter's command line.

4.1.2 Instrumenter's Options

`ckcc` accepts C/C++ compiler and linker commands, as well as its own specific set of options. The general call syntax is as follows:

```
ckcc [options] <cc-args>|<ld-args>
```

The ckcc-specific options are as follows:

-- time-locked Specifies that the source code contained in the file must be run at no time charge. In this mode, node switches cannot occur, and only the current context progresses. By default, this mode is disabled and every instruction executed is charged for a time quantum, allowing all nodes to be scheduled by the simulator.

-- net-shared Disables vectoring of non-automatic variables defined or declared in the source file. This means that all global variables found and data in otherwise static storage will not be prepared for multi-node simulation, leaving all variable in their native form. By default, vectoring is enabled.

--dry-run Causes the source code to be instrumented without performing the final compilation. This option is useful for validating a module's syntax without generating the corresponding object file.

-- stdout Processes the source file(s), whose instrumented version(s) are sent to the standard output without calling the compiler.

-- minimal Reduces the internal information added to the original source code by the instrumenter. This option currently disables all information for detecting reference conflicts between per-node and native variables while the simulator is running (see vectoring).

--c-ext=<.c>

--c++-ext=<.cc,.C,.cxx,.c++,.cpp>

--obj-ext=<.o>

These options specify the extensions for a C source file, C++ source file and object to the instrumenter, respectively. For instance, these options are useful when the application's files are maintained by a revision control system that uses non-standard file extensions. The bracketed values are predefined by ckcc.

--cc=<cc-command-line>

Specifies the command line used to call the C/C++ compiler for generating an intermediate object file or the final executable. To generate an object file, the default values are:

- “gcc for a C source file.
- “g++ for a C++ source file.

To generate a final executable, the default values are:

- “gcc” to edit links for a program consisting only of C object files.
- “g++ to edit links for a program containing at least one C++ object file.

--cpp=<cpp-command-line>

Specifies the command line used to call the C/C++ preprocessor for generating the intermediate source file the instrumenter will work on. The default value is obtained by appending the -E option to the current compilation command line value. The resulting instrumented file will then be passed to the selected compiler to get the corresponding object file.

--use-ck-cc Tells ckcc to use the C/C++ instrumenter as the final compiler. When applicable, this option must be specified during compilation and

link. (The GCC/EGCS version modified to include the CarbonKernel instrumentation generator still possesses all of its original capabilities as a regular C/C++ compiler).

--with-gnu-ld Tells `ckcc` that the current C/C++ development process uses the GNU linker. This option is generally not useful, as `ckcc` automatically determines the type of linker used. When specified this setting causes `ckcc` to pass the option `-Wl,-export-dynamic` to the linker.

--no-gnu-ld Forces `ckcc` to assume that the GNU linker will not be used, preventing the `-Wl,-export-dynamic` option to be passed.

--cplusplus Forces `ckcc` to assume that links are being edited for a program containing C++ code. As a result, the C++ linker (e.g. `g++`) is selected from the available default values. If `g++` is chosen, the `libstdc++.a` STL library will also be automatically included unless otherwise specified (i.e. `-nostdlib`). A command for integrated code generation (compilation then link in the same command line) automatically selects C++ mode if at least one C++ source file is detected in the modules to be compiled. This option is ignored if a `--cc` instruction forces the linker invocation syntax.

--rtos=<osX,osY,...> When links are being edited, this option specifies the list of executive models used in the application. It enables `ckcc` to automatically identify the RTOS model libraries that must be attached to the simulator. For instance, specifying `--rtos=ecos` links in the `eCos` kernel simulation model available from the shared library file **libecos.so** on a Linux platform.

--temp-dir=<temp-directory>
Specifies the access path for the directory that will contain the temporary files generated by `ckcc`. The default value depends on the host system. The temporary files will be deleted after processing, unless the option `--save-temps` appears in the command line.

--save-temps Forces `ckcc` to keep the generated temporary files. These files are produced during processing by the preprocessor as well as by the original files instrumenter.

--verbose This option causes the compilation and link commands issued by `ckcc` to be sent to the standard output.

--version Displays the instrumenter's identification version number.

--gcc-version Displays the version identifier for the instrumenter's host compiler.

--help Displays the list of options for the command.

<cc-args> Lists the arguments that will be passed unmodified to the compiler.

<ld-args> Lists the arguments that will be passed unmodified to the linker.

4.1.3 Locally Controlling Preemption and Vectoring.

In certain situations, it is important to block the instrumenter's action on a local level. The instrumenter's command line options `--time-locked` and `--net-`

shared, act upon all of the source file's content. Both have a local equivalent that only acts upon part of the code.

- ▣ Time progression can be blocked for a part of the code between the special service calls `CK_TIME_LOCK()` and `CK_TIME_UNLOCK()`. A lock counter runs between the two calls, which must be used symmetrically. The above two macro-definitions generate neutral code when the instrumenter is inactive, i.e. during the target code generation phase.
- ▣ Vectoring of a specific variable can be prevented by adding the `netshared` qualifier to its type declaration. This qualifier shows that the listed variable(s) is/are shared between all nodes, i.e. is/are unique throughout the scope of the simulation network. By default, this qualifier is implicitly selected for all variables declared in a system header file.

Care must however be taken to ensure that definitions and declarations for a variable used in several different files are consistent. For example, two files `fileA.c` and `fileB.c` use the global variable *Counter* and are compiled as follows:

```
fileA.c          fileB.c
int Counter;     extern int Counter;
```

```
$ ckcc -c fileA.c
```

```
$ ckcc -- net-shared -c fileB.c
```

The executable produced by linking in these object modules will be unusable, as the *Counter* variable in file `fileA.c` will be vectored whereas file `fileB.c` will create a traditional external reference to an integer (the result of using the `--net-shared` option).

The following example deactivates all vectoring of the two defined variables, so that they can be shared between all nodes. In this way, memory regions can be created that are shared between the different node contexts.

```
netshared char sharedMemory[SHMSZ];
netshared int locked;
```

The next example prevents the simulated time from progressing while the framed expression is running. This has two immediate effects on how the simulation runs:

- ▣ The current node cannot be preempted by another node.
- ▣ The expression protected in this way runs at no time charge.

```
CK_TIME_LOCK();
rqtBuf->next = NULL;
*sharedMemory->qTail = rqtBuf;
sharedMemory->qTail = &rqtBuf->next;
CK_TIME_UNLOCK();
```

4.1.4 Source Code Adaptation Requirements

Using CarbonKernel to simulate an application may lead the user to specially modify or organize the application source to be simulated and where necessary testing the instrumenter's signature to identify the current build context (i.e. target or simulation).

- As the processor targeted by the application code is generally incompatible with the workstation's local processor, and given that CarbonKernel is not a processor instruction set simulator but rather a RTOS simulator, inline assembly code aimed at the target processor cannot be simulated.

For this reason, no piece of application code written in assembler can be used without modification and all literal physical address references within application code are illegal; they will almost invariably cause a fatal error in the simulation process.

All interrupt handling routines, hooks or callouts must therefore be coded in C/C++.

- The *main()* function for running the application already exists in one of the CarbonKernel system libraries and so must not be redefined in the application code. For example, the *main()* function's name could be conditionally changed in the simulation's context, by testing to detect the CarbonKernel instrumenter's signature :

```
#ifdef _CKCC_
<simulator code>
#else /* !_CKCC_ */
<target code>
#endif
```

- The executable file must always be able to access its symbol table while it is running; when initialized CarbonKernel searches dynamically for the addresses of all functions linked to events, as well as the startup routine's address declared for each node. In other words, never strip out the symbols from a simulator's namelist.
- The instrumenter does not handle the conflicts raised when an attempt is made to aggregate several functions or data having the same name in a single simulation executable. These must be handled individually when the source code is adapted (e.g. identical variable or function names in two pieces of application code that will be run simultaneously on two different nodes). However, multiple instances of a piece of code can be run via the *ckcc* preprocessor.
- In C++, global references cannot be vectored and so must always be shared by all nodes, just because one cannot have pointers to C++ references (e.g. They are implicitly *netshared*).

4.1.5 Preprocessor Signatures

ckcc defines the CPP symbols “_CKCC_” and “_CARBONKERNEL_” before passing the source file to the C/C++ preprocessor. Those signatures can be used to have conditional compilation control over any pieces of source code that are reserved for running on the target or with CarbonKernel.

4.2 Ongoing Example: IBC Example

4.2.1 Introduction

Throughout this document, we shall refer to a sample simulation, which we will use as an ongoing example. The example, called IBC, uses two nodes, each of which runs a separate piece of application code requesting services to a simulated eCos kernel. This example is intended to show CarbonKernel's ability to simply and effectively simulate a distributed system environment.

Node "Server" receives requests via a faked shared memory region implemented by a regular non-vectorized C struct, and sends an acknowledgment for its running to the calling Client node. The system for signaling incoming requests and acknowledgments is based on using simulated hardware interrupts. The "Client" node periodically submits requests to the server and awaits a synchronous acknowledgment for each request.

In addition, a simulated manual interrupt can be used to cause requests to be sent from the CarbonKernel's graphical monitor to the Server on behalf of the Client node. This mechanism shows very simply the possibilities of injecting events into the simulation.

4.2.2 Distribution

A compressed *tar* archive can be found in the directory `src/ck-version` under the installation root directory. This archive contains the IBC example among others, which are pre-compiled in the `demos` subdirectory of a standard CarbonKernel distribution. IBC consists of four C files: `server.c`, `client.c`, `shm.c` and `shm.h` that will be compiled to build the simulation executable.

Once the archive contents are deflated and extracted, you should be able to run the `configure` script from the top-level directory. This is a GNU *autoconf* script which is aimed at creating the Makefiles needed to build the examples. `installdir` is the CarbonKernel installation directory.

```
$ ./configure --with-ck-dir=<installdir> --prefix=`pwd`
```

We will describe how to define a simulation architecture for this example later in this document.

4.2.3 Generating the Executable

The code generation rules are given by the Makefile file. Simply enter the following command in the directory containing this file:

```
$ make install
```

A simulation executable file called **ibc** is then created in the *demos* subdirectory.

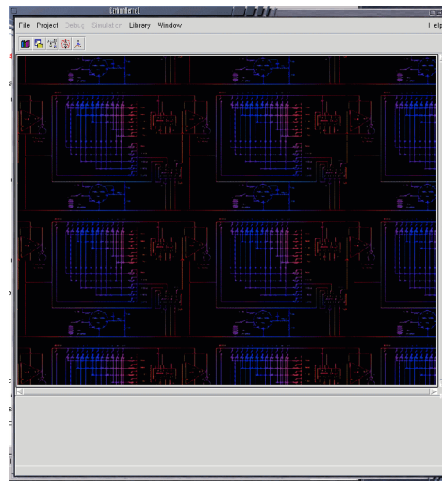
5. Configuring the Simulation

The second of the two main stages needed to set up a simulation consists of defining the simulated configuration (aka architecture). This step is controlled by a central graphic application combining all simulation session configuration and operation services. This application is known as the ISE, which is an acronym for Integrated Simulation Environment.

To run the ISE, start the following command:

```
$ ck &
```

The ISE initially appears as follows:



with a main toolbar, giving shortcut accesses to the following functions, from left to right, respectively :

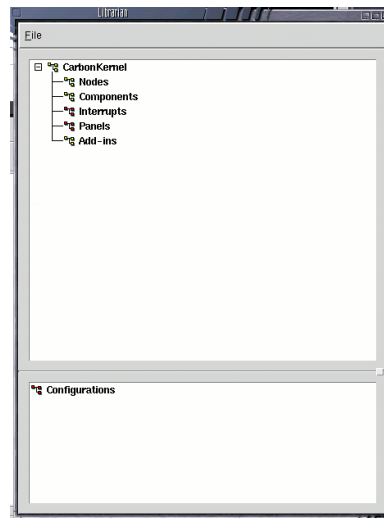


- ▣ Open the CarbonKernel's librarian to create/edit the simulated models instances.
- ▣ Select an existing simulation project file.
- ▣ Access the current project's options.
- ▣ Start the application under the integrated debugger's control.
- ▣ Start the application under the standalone monitor's control.

5.1 The Librarian

The Librarian is used to define and associate the various simulated elements that will allow the application code to run, e.g. the simulated RTOS characteristics and

settings. In the ISE, select the **Open** command from the **Library** menu, or click the icon (to the left of the toolbar). The following screen is then displayed.



This window shows the known models hierarchy, as well as the different existing instances. Two separate directory structures can be seen:

- ▣ The first, at the root of which is the **CarbonKernel** element, organizes instances according to whether they belong to the simulated targets (**Nodes**), interrupt generators (**Interrupts**), software components associated with a RTOS (**Components**), graphic panels (**Panels**), or FROGS native simulation models (i.e. **Add-ins**) class. In our example, we will define two targets each running an eCos kernel, as well as an interrupt source on one of the two nodes.
- ▣ The second, at the root of which is the **Configurations** element, lists the existing simulation configurations as a single level.

5.1.1 Simulated configurations

Once the various local settings of useful model instances have been individually set up, they must be associated together before the actual simulation can begin. The result of associating them together is called an *architecture or a simulated configuration*. The simulator produced by linking together the application code and the simulation kernel is then run in the environment described in this way.

The expression simulation node refers to the modeling of an embedded electronic board associated with a specific RTOS. The term target, node or simulated board will refer to that concept throughout the rest of this document.

A simulated configuration may be designed as a super-model instance, whose role is to associate the various elements of a simulation system to define its initialization rules. That process consists of five main phases:

- ▣ Creating and configuring the nodes (for example, creating our IBC example's **Client** and **Server** nodes; see below);
- ▣ Creating and configuring the interrupt sources found on the nodes by creating **Interrupts** model instances that must then be attached to the proper nodes (i.e. **CarbonKernel/Interrupts** heading in the node's configuration window). An instance of the target's real-time clock is

automatically created by the **eCos** model, thus not needing to be externally defined;

- Designing the graphic panels (for displaying the simulation's data inside Tcl/Tk widgets) by creating instances of the **Panels** model that must then be attached to the node (**CarbonKernel/Control Panels** heading in the node's configuration window);
- Creating and configuring instances of the **FROGS** native simulation models if any (there are none in our example);
- Setting up the interrupt generators, which means programming the interrupts generation law, e.g. Periodical, exponential, manual and so on.

5.1.2 Typical Architecture

In its simplest form, a **CarbonKernel** simulation consists of an architecture comprising a single target board, i.e. A single-node system. The IBC example offers a more complex architecture, which includes two nodes exchanging messages.

All information on the different known architectures is stored in a library file containing the simulation models and configurations. Only one library is active at any time within the ISE, and is only updated via the Librarian.

5.1.3 Loading Simulation Modules

RTOS models and more generalized **FROGS** native models are contained in shared libraries on the host system called `modules`.

These modules must be explicitly loaded by the Librarian before the models they contain can be instantiated. Each shared library or module export one or more implementations of simulation models, along with their configuration characteristics. The Librarian makes use of all that information in order to offer a suitable graphic configuration interface when instantiating these models.

The **CarbonKernel** Librarian's role is therefore to offer a consistent graphic interface for creating and configuring the chosen instances according to the required characteristics of behaviour. For example, an **eCos** simulation node may be tuned according to the kind of scheduler used (bitmap or multi-level queue) or the number of authorized scheduling priorities.

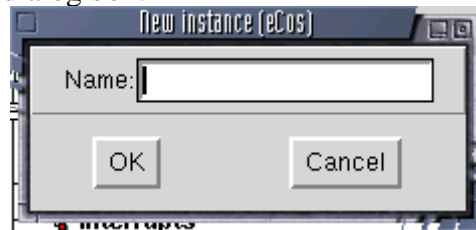
The various instances of models are organized within a library of simulated components. Using this approach, they can be reused in different contexts simply by association.

A module can be preloaded when the Librarian is first run under a given user account, using an automatic loading system triggered when a file named **autoload** is found in the **share/ck** subdirectory under the **CarbonKernel**'s installation directory.

This text file lists certain external resources that can be attached dynamically to the ISE. Among these resources are simulation modules which designates shared libraries containing simulation models. Each simulation module begins with the **cfmod=radix** followed by the library's generic name. For example, the **eCos** module is loaded by the expression **cfmod=ecos**. The corresponding module file can be accessed from a regular installation's root directory, in the area reserved for libraries (e.g. **lib/libecos.so** on a Linux platform).

5.1.4 Creating/Copying a Model Instance

- To create our example's **Server** node, right-click the **eCos** node in the models directory structure, and then select the **New** command to display the following dialog box:



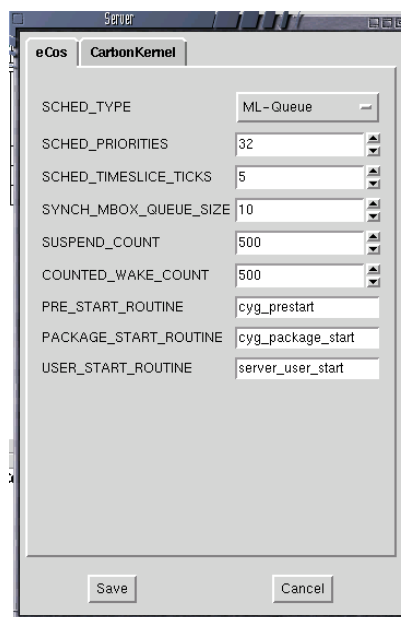
- Type the new instance's name, for example **Server**, and then validate with the **ENTER** key or click the **OK** button. Repeat the same process for the **Client** node. After each new instance is created, CarbonKernel automatically opens it for editing.

A node's configuration characteristics mainly depend on the simulated RTOS characteristics. Always refer to the simulation model's implementation documentation to see the meaning of the available settings.

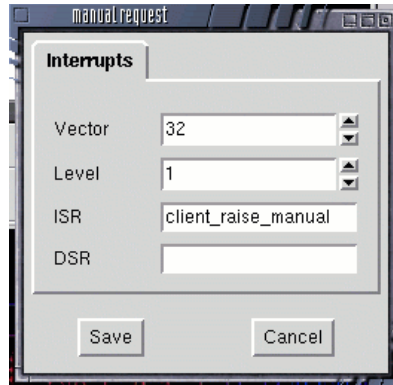
- To duplicate an existing instance, select the **Copy** command from the previous menu. A dialog box is then displayed, in which you enter the copy's name.

5.2 Modifying/Renaming a Model Instance

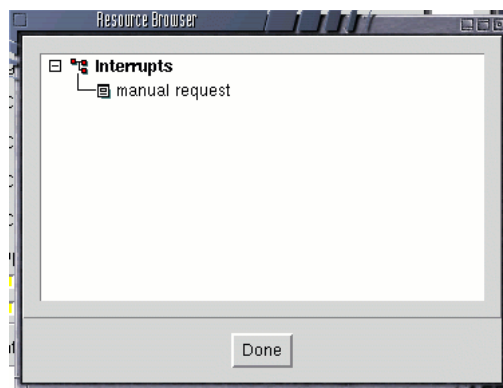
- Each instance of a simulation model exports a series of settings through which it can be configured. When an instance is created, a default set of values chosen by the model's designer is initialized. If you wish to modify those settings, double-click the selected instance's name, or right-click the icon representing it to display the local menu from which you select the **Open** command. If you use this command on the **Server** node instance that you have just created, the following configuration window is displayed:



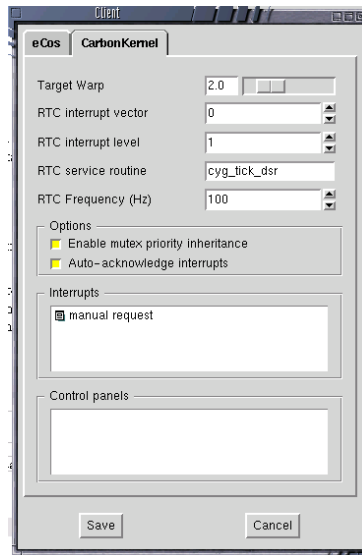
Each tab represents a specific category of the node's settings. The **eCos** configuration notebook displays an **eCos** tab for the kernel settings and a **CarbonKernel** tab, containing a set of simulator-specific settings. In our example, an interrupt source must be defined, that we shall call *manual request*, so that a manual request can be triggered. To define it, create an instance of the **Interrupts** model found under the **CarbonKernel** root, with the following settings:



- After validating the interrupt creation by clicking the **Save** button, open the Client instance's, switching to the **CarbonKernel** tab. Right-click the list of attached interrupts (i.e. Interrupts heading), and then select the **Insert** command in the local menu. This function is used to select the hardware interrupts that will be simulated on the node. The list of known interrupts is then displayed:



- Double-click the *manual request* instance to make the association and then end by clicking the **Done** button. In our example, the **CarbonKernel** tab appears as follows once the association has been made:



When all the settings have been correctly updated, validate the modifications with the **Save** button.

- If you wish to rename a specific instance, right-click the icon representing it in the models directory structure, and then select the **Rename** command in the local menu. A dialog box is then displayed, in which you enter the instance's new name. Renaming an instance does not affect any existing links between the different instances.

5.3 Destroying a Model Instance

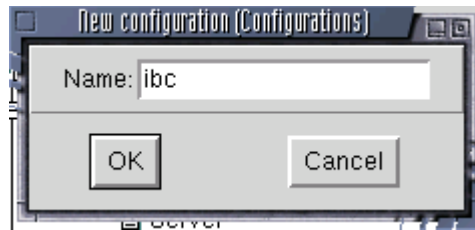
If you wish to delete a model instance, right-click the icon representing it in the models directory structure, and then select the **Delete** command in the local menu. A dialog box is then displayed, asking you to confirm the action.

Deleting an instance automatically destroys links from other model instances to the destroyed item. That/those model(s) can therefore no longer work correctly if those links must exist.

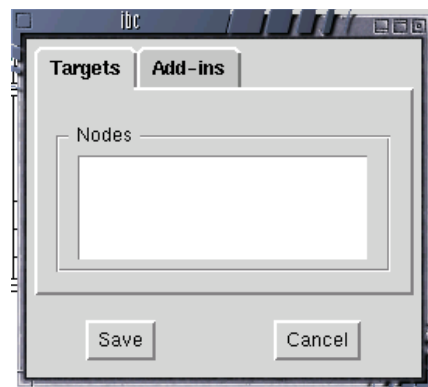
5.4 Defining a Simulated Configuration

5.4.1 Associating Nodes

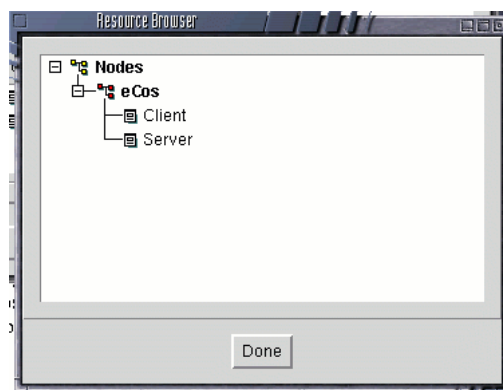
A simulated configuration or architecture is an association of different simulation models instances, including RTOS instances, that are intended to work together within the CarbonKernel simulator. To create a configuration from the CarbonKernel Librarian, right-click anywhere in the second directory structure called **Configurations**, and then select the **New** command in the local menu. A dialog box is then displayed, in which you enter the new configuration's name, e.g. `ibc`:



Validate with the **ENTER** key or click **OK** to continue creating the configuration. When it has been created, the new configuration is displayed so that its characteristics can be defined. The following window is displayed:



The graphic representation of a configuration displays two tabs: one for nodes (i.e. **Targets**), and the other for FROGS native simulation models (i.e. **Add-ins**). The first phase consists of choosing the nodes. Right-click the nodes list (i.e. **Nodes**), and then select the **Insert** command in the local menu. This function is used to choose the nodes that will be associated together in the simulation. In our example, the displayed selection window appears as follows:



Double-click both **Client** and **Server** to include the two nodes you have just created. In our example, the order in which they are selected is unimportant. When you have done so, click the **Done** button.

Each node will be allocated a unique numerical identifier when the simulation is run. This identifier (called **nid** for **Node Identifier**) is allocated starting from 0 for the first node appearing in a configuration's nodes list and is then incremented by 1 in order of the nodes declarations in the list.

5.4.2 Linking Add-ins to the Simulation

When Add-ins are needed to set up a simulation, select the **Add-ins** tab and then continue as before to make the association.

An Add-In can for example simulate the behaviour of an 802.3 type network linking the multiple nodes together. Such kind of model is useful in refining a configuration's behaviour. In simpler cases, only trivial routines that very roughly simulate message transmissions may be needed. You must analyze your simulation requirements to decide whether to include an Add-In or not.

Our example does not involve any Add-ins.

6. Simulation Project

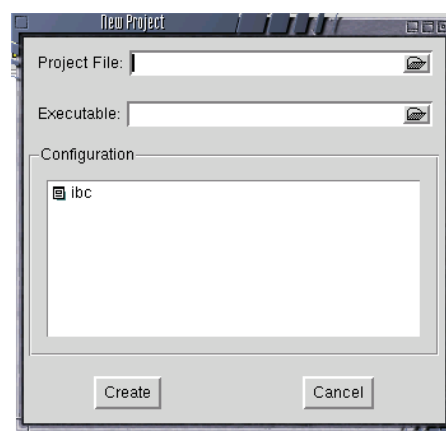
A CarbonKernel simulation project is created by associating a simulated configuration with a simulation executable.

All information regarding a specific project is stored in a separate file, which refer to the model library that was active when the file was created.

Before making the first simulation of the IBC example, that association must be made using the **Project** menu's commands.

6.1 Creating a Project

Open the **Project** menu, and then select the **New** command. A window is then displayed, offering two selection fields: one to select the new **Project File**, and the other to select the associated **Configuration**.



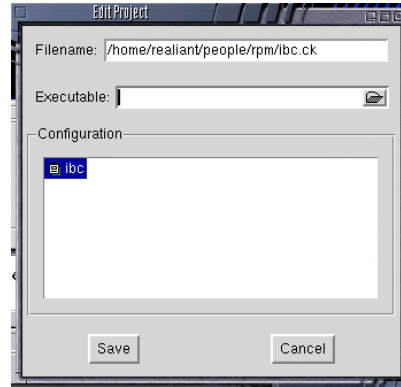
- ▣ In the first field, enter the project file's access path. If you wish, you can use a browser by clicking the icon displayed to the right of the input field. The file must not already exist. CarbonKernel automatically adds the **.ck** extension, which refers to all simulation projects' filename.
- ▣ You can enter the simulator's file name in the second field. Environment variables may be used to represent all or part of the access path, and will only be expanded when the ISE actually uses the access path. Note that the simulator's file name need not be input at this point when the project's initial details are entered. In that case, the project must later be modified to complete its definition once the simulation executable is specified.
- ▣ Select a simulated configuration in the displayed list, by clicking its name so that it is highlighted. Our example configuration **ibc** should appear in this list.
- ▣ After you have specified all the above information, click the **Create** button to create the project.

The new project is automatically selected by the ISE.

The ISE's commands for running a simulation can only be accessed when a current project has been selected and the project's settings specifies a valid simulator's executable file and simulated configuration.

6.2 Modifying a Project

After selecting the current project, select the **Edit** command from the **Project** menu. A window is then displayed, offering two input fields: one to select the simulator's **Executable**, and the other to select the associated **Configuration**. The current project's filename appears at the top of the dialog box, in read-only form.



- ▣ Enter the access path and name of the simulator's executable file in the field labeled **Executable**. If you wish, you can use a browser by clicking the icon displayed to the right of the input field. The selected file must be a simulator's executable generated separately by the normal **CarbonKernel** procedure. Until a valid executable file has been specified in this field, all the ISE's commands for running the simulation are inactive. This field may contain environment variables that will be evaluated when the access path is actually used.
- ▣ Select a simulated configuration from the proposed list by clicking its name to highlight it. In our list, we will find our example configuration **ibc**.
- ▣ When you have selected the previous two items, click the **Save** button to modify the project.

6.3 Selecting a Project

To change the ISE's current project, select the **Open** command from the **Project** menu. A browser is then displayed, with which you can select the chosen project file. Double-click the filename displayed in the browser, or select the file with the right mouse button and then click **OK**. From then on, all the ISE's commands will act upon the new active project, whose name is displayed in the main window's title bar.

The most recently opened projects can be quickly accessed using the **File** entry in the **Projects** menu.

7. Control Panels

Control panels are graphic windows used to display a set of interactive controls called *magnets*, which show the simulation variables' contents. A control panel is attached to a specific node. There is virtually no limit - except readability — to the number of control panels that can be associated with any node, or the number of *magnets* that can be displayed by any control panel.

7.1 Message ports

Each graphic control displayed on a control panel receives its data from an event bus managed by the simulator. The access point for sending and receiving information on that bus is called a *message port*. As the simulator's activities send information on the event bus, that information is automatically sent to the defined monitors, and especially ISE which then channels it to the appropriate graphic controls.

7.2 Dataports

A *dataport* is a special type of message port, because it is associated with a program variable in the language used to write the application. In other words, each change in a variable associated with a *dataport* is automatically signaled on the event bus. A graphic control monitoring a *dataport* therefore shows a program variable's current value and guarantees that it is updated in real time. In C/C++, a *dataport* can be distinguished by the ***dataport*** qualifier in the variable's declaration syntax.

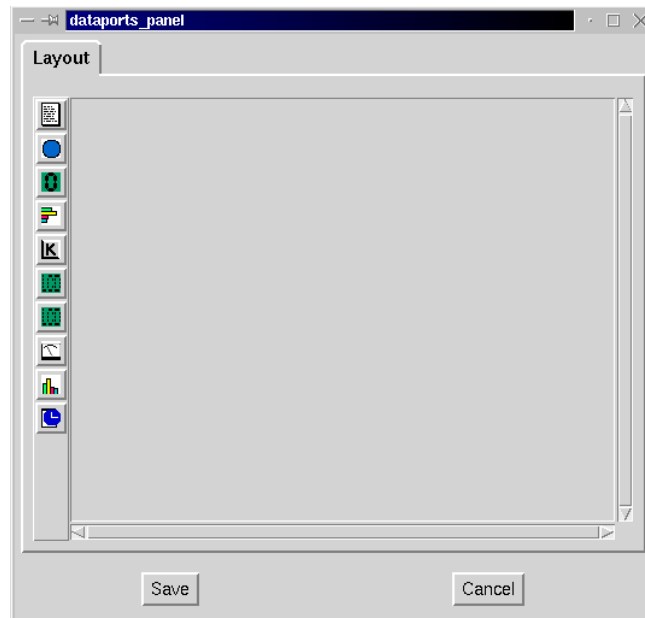
7.3 Magnets

Magnets are independent graphic objects that are grouped together within control panels. They are *Tcl* language modules with a standard interface defined by the ISE so that they can receive notifications from the event bus and if necessary resend information on the bus. New *magnets* can therefore be included within the ISE. For this, they must obey the public interface standards documented in the **CKPI** document's "Writing animated magnets" chapter.

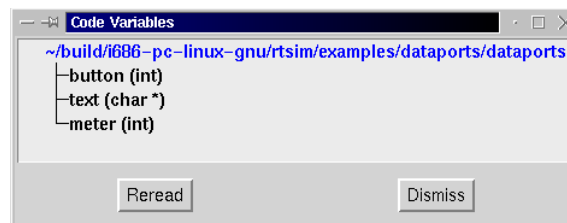
7.4 Creating Control Panels

We shall create a control panel for the **magnets** example. The example updates certain *dataports* at regular intervals.

- ▣ Open the Librarian and create a new **eCos** node. Call it "eCosNode". Validate by clicking **Save**.
- ▣ Create a new configuration called "MagnetConfig" and attach "eCosNode" to it.
- ▣ Create a new project using the **magnets** executable found in the **demos** directory under the installation's root directory, together with 'MagnetConfig' configuration.
- ▣ In the Librarian, right-click **Control** and then select the **New** command in the local menu to create a new control panel. Call the control panel ""SamplePanel"", and then validate by clicking **OK**. An empty panel is then displayed:



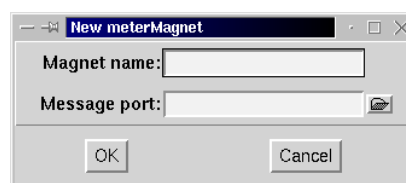
A set of icons is displayed on the panel's left side. The first icon opens a window listing all of the defined *dataports*:



The simulator's executable's filename is displayed at the root of this tree structure; the items in the tree structure are the names of variables declared as *dataports*, together with their respective C/C++ data type.

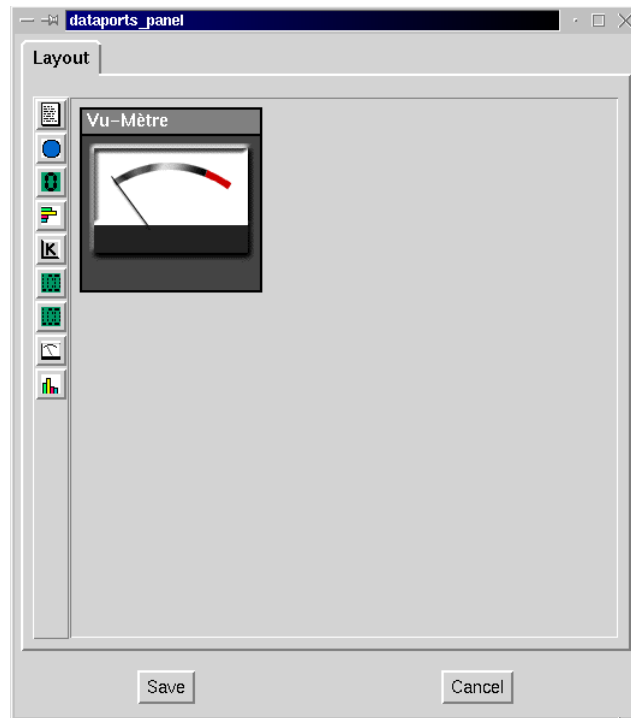
All the other icons displayed on the panel's left side are the available *magnet* types. We will now add a gauge type *magnet*.

- ▣ If necessary, close the *dataports* window. Click the icon representing a gauge, to the left of the "SamplePanel" window. The window which is then displayed is used to define the *magnet's* name and link it to the simulation via a *message port*:

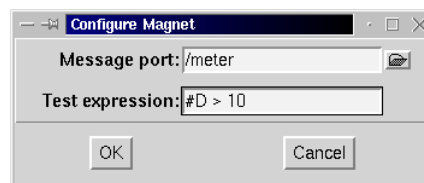


The icon to the right of the second field is used to choose the *message port* from the list of available *dataports* (remember, a *dataport* is a special type of *message port*). This list is automatically filtered so that it only shows the *dataports* compatible with the current type of *magnet* (in this case, a gauge).

- ▣ Call this *magnet* "'Meter'", and then click the *message port* icon to open the list of compatible *dataports*. Meter type *magnets* accept integer values. Select the meter" port, and then click **Apply**. Its name appears, preceded by the "/" character to show that this is a *dataport* and not a basic *message port*. Validate with **OK**. The *magnet* is then displayed in the control panel:



The *magnet* is decorated with a title bar, title and border. It is entirely manipulated via the title bar: the magnet can be moved using the mouse's left button; a local menu can be displayed by using the right button. In this menu, the **Rename** command is used to rename the *magnet* and the **Delete** command is used to remove it from the control panel. If the *magnet* was designed to be customizable, the **Customize** command appears in the menu and opens a window specifically for that type of *magnet*. The **Configure** command displays the following window:

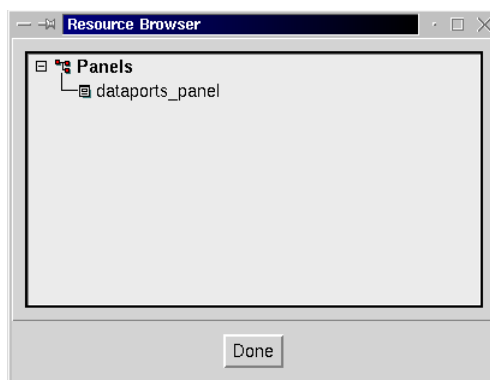


The *message port* used by the *magnet* can be changed from this window, either by typing its name directly or by choosing a *dataport* in the list. A test expression can also be entered, that will be evaluated every time that the value of the *message port* associated with the *magnet* changes. The simulation is automatically suspended when the expression becomes true. The simulation can then be resumed using the ISE's **release** button.

The expression's syntax is described in the documentation for the **ckSetTestExpr()** service, in the **CKPI** document.

We will now associate the control panel with the "MagnetConfig" configuration that we created earlier.

- Save the control panel by clicking **Save**, and then open "eCosNode". Click the **CarbonKernel** tab, and then right-click the list of control panels attached to the node (i.e. **Control Panels**). Select the **Insert** command in the local menu. This function is used to select the control panels that will be attached to the node. The list of known control panels is then displayed:



- ▯ Double-click “SamplePanel”, and then click **Done** to close the window. Save these settings by clicking the **Save** button.

When the simulation is run, the gauge is activated and, because of the expression already defined, stops the simulator when the associated *dataport's* value is greater than 10.



8. Editing the project settings

Editing the current project settings consists both of choosing the laws governing how the different events defined for each node are generated, and making certain settings in the control tools or the simulator itself. All these settings are stored for the active project.

8.1 Configuring Event Sources

8.1.1 Events, Interrupts and the Simulation Scenario

A simulation scenario is defined by choosing laws governing how the events attached to the nodes of a given configuration are generated. CarbonKernel triggers an event when needed by calling a service routine attached to it. An interrupt is an extended event defined by the VRTOS whose service routine immediately raises an IRQ on the current node. The ISR and optionally the DSR attached to the interrupt can be configured when editing the corresponding interrupt model instance.

CarbonKernel offers seven predefined generation laws, each of which defines how frequently an event is raised. Configuring an event source consists of setting a selector and entering a generation law setting. The following summary lists the types of predefined event sources, with the graphic selector position and corresponding setting syntax for each. Both settings should be provided for each source appearing in the configuration's **Events** window, otherwise the unconfigured source remains inactive during the simulation.

If no event is defined for the current configuration, the **Events** window is not displayed.

8.1.2 Event Source Settings Syntax

Periodical	Periodic	[t0-tmax/]<period>
Exponential	Exponential	[t0-tmax/]<mean>
Uniform	Uniform	[t0-tmax/]<dmin-dmax>
File	Source file	<source file>
Manual	Manual	<destination file>
Timer	Unique	<time>
Null	-no call-	-no setting-

8.1.3 Generating Events Automatically

An automatic source is handled by the simulator without any action by the user being needed.

An automatic source relates to a domain **[t0-tmax]** representing the simulated time frame during which the source should be active. No event can be generated outside its limits. If no limits have been defined, the source is active throughout the simulation. A simulated time limit is expressed by an absolute time value, that may be associated with a unit, e.g. 100 msc (usc = microseconds, msc = milliseconds, sec = seconds). The default unit is the microsecond (usc). If the

tmax limit's units are not specified, it always uses the unit chosen for limit **t0**. The limits are separated by a hyphen, and separated from the rest of the setting by one of the characters “/”, “:” or “,”. The following syntaxes can also be used:

tx/...	tx-<simulation end>
-tx/...	<simulation start>-tx
tx-/...	tx-<simulation end>

- For periodical sources, the **period** setting is a duration representing the generation law period, expressed in simulated time. It is expressed in the same way as a simulated time limit.
- For exponential sources, the **mean** setting is a duration representing the mean of the generation law, expressed in simulated time.
- For uniform sources, the **dmin-dmax** setting is a domain expressed by two simulated-time limits representing the minimum and maximum limits of the generation law. If **dmax** is omitted, the domain used is equal to **[0-2*dmin]**.
- For file sources, the setting is the access path for the file containing generation start time instructions, with one start time in each line. So that the rest of the file is correctly interpreted, the first line must begin with the marker “**#\$@timelog**”. All other lines in the file are evaluated according to the same rules as a simulated time window limit, until an invalid character or end of line is reached (spaces and tabs are ignored). Any line beginning with the “**#**” character is seen as a comment; empty lines are permitted and ignored. For example, the following file extract lists three start times expressed in microseconds, followed by two others expressed in seconds.

```
# $@timelog
182.867 usc
183.1
184.9
12.5 sec
12.7 sec
```

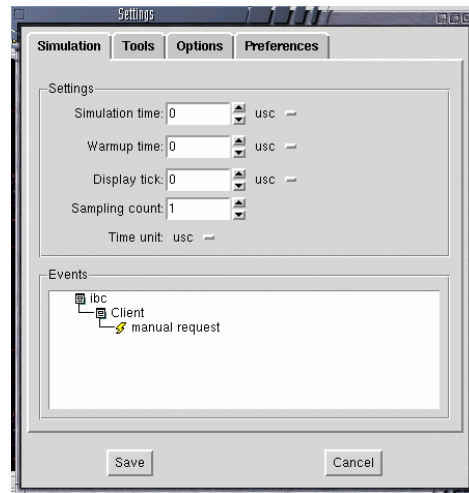
- The **time** setting in **Timer** mode follows the same rules as a simulated time window limit, and specifies a single fixed time for triggering the event.
- The **Null** position inhibits the source throughout the simulation.

8.1.4 Generating Events Manually

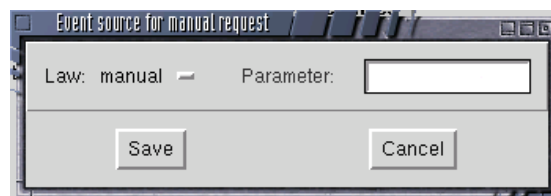
A manual source is linked with a graphic button that can be accessed via the simulation inspector. Each click on this button immediately triggers the event. For manual sources, the optional setting is the access path for an output file that

will record the list of event start times injected by the user, in a format that can be reused by an automatic source's file mode.

In our example, the manual request interrupt can be configured in manual mode. To do so, use the **Simulation** menu's **Configure** command. The following window is then displayed:



This window lists the events defined in the current configuration under the **Events** title, so that their generation laws can be set there. Double-click the icon representing the interrupt **manual request** so that the following dialog box is displayed:



Choose the **Manual** position without specifying any setting, and then validate by clicking the **Save** button.

8.2 General Simulation Settings

The ISE can be used to set a series of settings relating to the simulation's general behaviour. The parameters listed under the **Settings** heading of the configuration window discussed above mean the following:

- ▣ **Simulation time** is the simulation's duration. When the simulated time reaches this value, the monitor permanently stops the simulation. The value 0 is understood as an indefinite duration. In that case, the user must explicitly stop the simulation.
- ▣ **Warmup time** is the simulation's “warm-up time”, i.e. the startup period during which no statistical samples are taken. This period must correspond to the simulation's transitional initialization period, during which the statistical data is unimportant or inaccurate.

Warning: The total duration of the simulation is made up of the warm-up period added to the **Simulation time** period during which readings are taken.

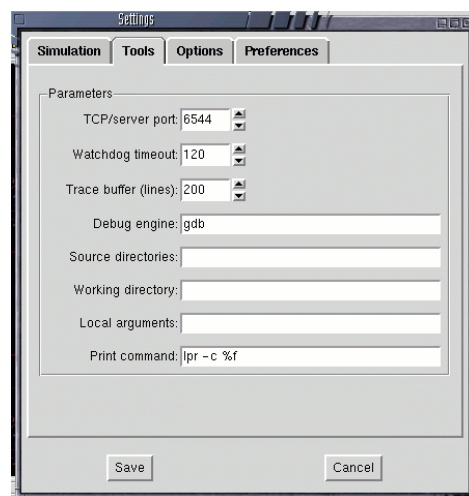
- ▣ **Sampling count** is the number of statistical samples that must be taken.

If the value is null, the simulation is considered as having an indefinite duration and the time indicated for **Simulation time** is taken as the sampling period. This setting's default value is suitable for standard simulations. This is an advanced setting that is only used for certain types of simulation that include highly specialized **FROGS** add-ins.

- ▣ **Display tick** defines the smallest time interval that can be displayed. For example, if the display tick value is 0.1 usc, all times will be displayed to the nearest tenth of a microsecond.
- ▣ **Time unit** is the default unit for displaying time. For example, if the value of **Time unit** is msc and the value of **Display tick** is 1 usc, all times will be displayed in the following format: 1 010.005 u. For example, if the value of **Time unit** is sec, the time will be displayed as 1.010 005 s.

8.3 Simulation Tool Parameters

A second series of parameters can be accessed via the main configuration window **Tools** tab. These can be used to modify certain parameters for the monitor function and debugger built into the ISE. The window is displayed as follows:

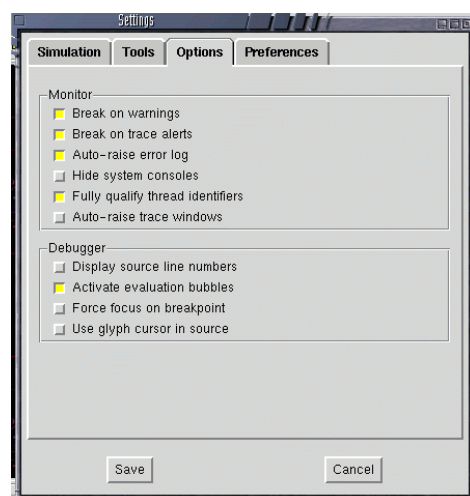


- ▣ **TCP/Server port** is the TCP/IP port number used for communication between the ISE's monitor and the simulator. The default value was chosen in order to avoid any conflict with the system ports. In addition, as this port only remains active for a very short period of time while a simulation is initialized, it can be shared by several users. Nevertheless, if any conflict arises this value can be changed using this parameter for the active project.
- ▣ **Watchdog Timeout** is the time limit before which the connection must be made between the monitor and simulator, in seconds. If this limit is exceeded the ISE simulation startup process is automatically interrupted and an error message is displayed.

- ▣ **Trace Buffer** is the number of lines of text that can be stored in the API's trace results window. A non-null value means that the oldest messages are removed once that number of lines is reached. A null value represents a trace buffer of unlimited size. The default value is 200 lines.
- ▣ **Debug Engine** is the identifier of the debug engine used by the ISE to control the simulator via the integrated graphic debugger. Various engines could be used provided a support library is available. The standard engine supported by CarbonKernel is GDB.
- ▣ **Source Directories** contains the list of access paths to the directories that contain the application's source files; that list is used by the integrated debugger. The expression used must be compatible with the syntax expected by the debug engine associated with the ISE. GDB is the default engine used. This is an optional parameter.
- ▣ **Working Directory** indicates the access path to the simulator's startup directory. This is an optional parameter.
- ▣ **Local Arguments** is an area specifically for simulator startup arguments that are local to the application. Ideally, these options begin with the reserved prefix (-Q) for this purpose, in order to avoid any conflict with the simulator's internal options. These options are always added unmodified to the end of the list of internal options set by the ISE when the simulator is run. See the *Command line* section for more detailed information on this subject.
- ▣ **Print Command** contains the command line that will be used to submit print requests to the host system. A shell interpreter is called to run the request. The first occurrence of the “%f” control string is replaced by the name of the file to be printed, if that string appears in the body of the command. In this way, the name of the file that must be printed can be positioned in the command line in the required syntax.

8.4 Operating Options

A final series of options can be accessed via the **Options** tab. These can be used to enable or disable certain of the simulator's and ISE's features. The window is displayed as follows:



The possible settings are:

- ▣ **Break on Warnings:** selecting this option causes a temporary break in the simulation when the simulator returns a warning. If the integrated debugger is running, the line of code that caused the error is highlighted. In addition to the temporary break, the message is repeated in the ISE's **Error log** window, which can be accessed via the **Windows** menu. Such warnings are generally produced by simulation models; however user code can also generate messages via a special service. For details, see the **ckWarn()** routine detailed in the CKPI documentation.
- ▣ **Break on Trace Alerts:** selecting this option causes a temporary break in the simulation when a trace alert is returned by the simulator that has the alert attribute set. Only applications can generate this type of alert, using the CKPI interface's **ckTrace()** service. Alerts of this type are stored in the API's trace manager window.
- ▣ **Auto-raise Error Log:** selecting this option causes the **Error log** window containing the simulator's alert messages to be automatically displayed in the foreground as soon as a new message is received.
- ▣ **Hide System Consoles:** selecting this option prevents system consoles from being created for nodes when the simulation is run. Consequently, the simulator's standard I/O streams are not redirected.
- ▣ **Fully Qualify Thread Identifiers:** selecting this option allows the long identifier to be displayed for threads in the focus selectors (debugger and traces). A long identifier attempts to display the name of the thread's entry point instead of its internal identifier. For example, `ibc_server(Server)` is the long identifier for the server thread running on the **Server** node in our IBC example, whose entry point is defined by the `ibc_server` C routine. If this information is unavailable, the internal identifier is displayed alone.
- ▣ **Auto-Raise Trace Windows:** selecting this option causes the inspector's, the memory examiner's and the global variables windows to be automatically raised in the foreground each time that the ISE has taken control of the simulation (e.g. code stepping, break).
- ▣ **Display Source Line Numbers:** selecting this option causes the source file's line numbers to be displayed in the debugger's frames.
- ▣ **Activate Evaluation Bubbles:** selecting this option causes expressions under the mouse pointer in a debug frame to be automatically evaluated. If you are working with simulators having huge namelists, you may consider deselecting this option to improve the ISE's responsiveness when moving the mouse pointer over the source window.
- ▣ **Force Focus on Breakpoint:** selecting this option causes the debugger's *Follow thread* function to be run whenever any break is caused by a breakpoint hit in the simulation. This automatically sets the focus to the context that was active when the simulator was suspended, so that that specific context can be tracked step by step later.
- ▣ **Use glyph cursor in source:** selecting this option replaces the highlight normally used to indicate the current line in the debugger's frame by a graphic cursor in the left margin. This mode is automatically chosen if a monochrome display is detected.

9. Exporting/Importing the work environment

The exact contents of a CarbonKernel work environment can be exported to a portable ASCII format and then imported again using the **Project** and **Library** menus' **Import/Export** commands. These functions can also be obtained in combination using the ISE's **-e** and **-i** command line options.

Unlike the ISE's command line functions, which produce a combined export file containing both the project information *and* the resources library contents, the functions accessed via the graphic menus concern either the project *or* the resources library contents.

9.1 Exporting/Importing the current project

Choose **Export** in the **Project** menu. After you have validated the export file's name, every item in the active project is recorded in the export file in portable ASCII format. You can import those items again using the same menu's **Import** function. The imported items permanently replace the active project's existing contents.

9.2 Exporting/Importing the models library

Choose **Export** in the **Library** menu. After you have validated the export file's name, every item in the resources library is recorded in the export file in portable ASCII format. You can import those objects again using the same menu's **Import** function. The active library will be incrementally updated with any missing or modified objects from the import file.

10. Running a Simulation

10.1 Interactive Execution

The CarbonKernel ISE offers two built-in simulation control modes. The first mode provides a display and control monitor for handling system objects present in the simulation. This mode also provides step-by-step functions that are synchronized with certain events such as executing system calls or changing thread status. The second mode adds a symbolic debugger to the previous mode. The functions of this debugger are extended to the CarbonKernel multi-focus control system. For example, a powerful focus management function lets you see the concurrent operation of a number of threads running on different nodes.


10.1.1 Simulation Monitor

An additional toolbar displays the functions available from the **Simulation** menu when the simulation is running. These functions will be described in detail throughout this section. When the monitor is in standalone control of the simulation, this toolbar's actions are the following, from left to right:

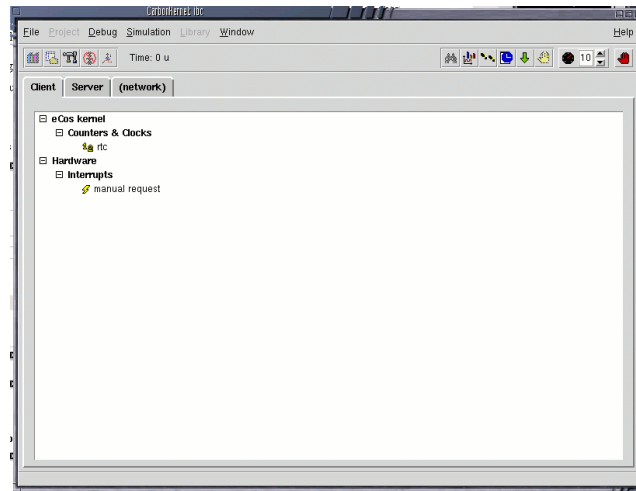


- ▣ Inspector button (disabled in this mode).
- ▣ Graphical plotter button.
- ▣ APIs tracer button.
- ▣ Simulation timers button.
- ▣ Continue simulation.
- ▣ Stop simulation (disabled).
- ▣ Host simulation speed selector.
- ▣ Iconic state information.

10.1.1.1 Starting Simulation

Starting the simulator using the monitor alone is controlled by the **Run** command in the **Simulation** menu. In addition, the  icon in the toolbar is a direct alias for this command. After starting the simulator, the list of system objects is displayed, arranged under different headings. There is one heading for each node, and one cross-functional heading that groups general objects that cannot be assigned to an individual simulation node. This heading is called **network**.




In our example "ibc", the following display is generated:



This presentation immediately displays the following information:

- Two nodes, **Client** and **Server** respectively have been instanced for this simulation.
- An eCos counter object underlying the **Client** node's real-time clock has been created.
- A manual interrupt has been defined on the **Client** node called **manual request**.
- Both nodes have no other system objects created yet, because the simulator has entered its initial break state at the beginning of the first (i.e. **Client**) eCos node's entry point.

10.1.1.2 Controlling Simulation











- Simulation is automatically suspended at time value "0" after general simulator initialization. Select the **Release** command from the **Simulation** menu to start execution. This command has an alias in the form of the  icon in the simulation control bar displayed on start-up in the right of the window. You can use this command to resume simulation after any suspension, regardless of the reason (breakpoint, step-by-step mode, etc.). After resuming the simulation, the current simulated time shown under the **Time** label in the monitor resumes incrementing.
- To suspend simulation again, select the **Hold** command from the **Simulation** menu or click on the  icon in the control bar.
- Host simulation speed can be controlled using the  selector. Slowing the simulator process is achieved using the activity on an internal hog thread whose sole function is to periodically suspend the simulation process. The maximum simulation speed is obtained when the selector displays a value of 10. At the other extreme, a value of 1 corresponds to the lowest speed. The host simulation speed should not be confused with the *Target Warp* discussed earlier. It does not apply to the simulated nodes, but rather to the simulation process. In other words, changing its value has effects in the actual time, not in the simulated time.
- To definitively stop simulation, select the **Kill** command from the **Simulation** menu.

- ▣ To restart the simulation using the monitor alone or with the debugger, use the **Restart** command from the **Simulation** or **Debug** menus as appropriate.

It is not necessary to suspend simulation in order to use the monitor tool functions, especially the simulation inspector.

10.1.1.3 Simulator Status Indicator

The current simulator status is shown by an indicator displayed at the extreme right of the monitor toolbar. The pictogram displayed specifies whether or not simulation is active, and in the latter case, why it was suspended. Given the large number of conditions that may warrant simulator suspension, it is useful to refer to this pictogram to determine the type of suspension. The following table summarizes possible status displays:

	Unconditional suspension
	Suspension caused by receiving a warning message
	Suspension caused by a timer that went off
	Suspension caused by receiving a trace message
	Suspension linked to a breakpoint activated in the graphs
	Suspension caused by a code breakpoint <i>(debugger)</i>
	Suspension caused by a data watchpoint <i>(debugger)</i>
	Suspension caused by a fatal exception <i>(debugger)</i>
	End of simulation
	Simulation in progress

10.1.1.4 Using the Inspector

The inspector is an important tool proposed by the simulation monitor. Its function is to present the system objects created and altered as the application code is run. It allows interaction with these objects depending on the rules defined by the designer of the simulation models used. The central graphic window in the monitor mode is used by the simulation inspector. In debugger mode, a dedicated window is created and is accessed using the **Inspect** command from the **Simulation** menu.

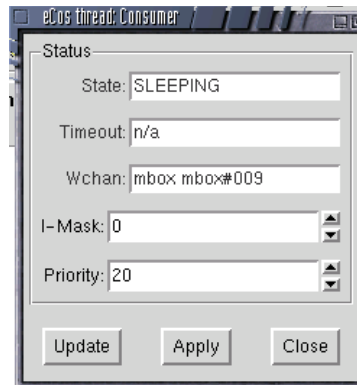
10.1.1.4.1 Object Organization

CarbonKernel organizes system objects using a hierarchy specified by the simulation model designer. A heading exists for each node where all of the system objects that are exported are grouped in hierarchical order. An additional heading called **network**, groups all of the general objects that

are globally assigned to the simulation network and that are not specifically dependent on an individual node.


10.1.1.4.2 Interaction with an Object

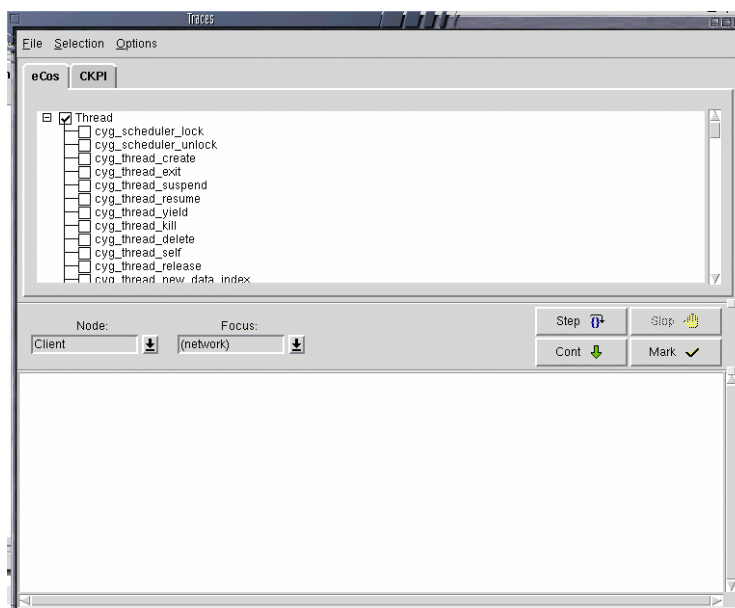
Each object presented has its own graphic representation that is used to display and possibly change its status and/or its characteristics. Let the simulator run for a short while using the "Continue simulation" green arrow icon, then hold it again using the "Stop simulation" yellow hand icon. Double-click on the icon for the **Consumer** thread in the **Client** node to call-up the interface shown below:



The possible interaction with each object exported by a given simulation model is dependent on the model implementation itself. For an eCos thread, you may change the interrupt mask and the scheduling priority interactively.

10.1.1.5 Tracing the kernel APIs

The CarbonKernel monitor has a system calls trace function that is used by the application. Click on the  icon in the simulation control bar or select the **Call interfaces** system object from the inspector's **network** heading. In our example, the following window is displayed:



This window displays all of the known kernel APIs, displaying one interface per tab. Each tab lets the user access the designated interface. The trace output buffer is located in the lower portion of the window. Traceable system calls are grouped by families. Check the routine names in order to get them traced.

The **Node** and **Focus** selectors let the user specify a reference context for tracing system calls. Any call made outside of the selected context will not be logged in the trace buffer. On the other hand, calling up a traced service in the selected context will generate a trace message in the trace buffer.

The **Node** selector lists known nodes. The **Focus** selector shows those threads which are active on the node chosen by the **Node** selector, plus two generic inputs called **(network)** and **(node)**. The former represents the global context, regardless of the node selection. In this case, traced calls will all be reported, regardless of the context on behalf of which they were invoked. In this mode, you gain an overview of all system calls invocations, over all contexts of all nodes. The second input represents all of the synchronous (threads) or asynchronous (ISR/DSR routines) contexts for the selected node.

The **Selection** menu is used to apply a global operation to all trace points:

- ▣ use **Select All** to select all trace points in a single operation,
- ▣ use **Select None** to inhibit all trace points in a single operation.

The **Options** menu provides an additional range of settings:

- ▣ use the **Trace Callouts** selector to include context change information in the active trace. This information includes thread creations, deletions and changes.
- ▣ the **Break on error** selector holds the simulation when a traced system call sends back an error state.
- ▣ the **No filtering** selector is used to inhibit filtering on system call traces. In other words, this action is the same as selecting all of the calls present in the various trace selectors. Focus is applied. This option implies step-by-step execution mode.
- ▣ **Hide selectors** masks or unmasks the system calls selection list. This option is useful for dedicating the entire window display area to the trace buffer, once the system calls to trace have been selected.

The following four buttons with pictograms allow the user to trace operations:

- ▣ **Step** starts the step-by-step mode, where simulation is automatically held after each trace is received.
- ▣ **Release** releases the simulation by canceling, where applicable, any previous step command. This command is almost the same as the one in the **Simulation** menu, except that it continues the trace using the current selections.
- ▣ **Hold** unconditionally holds the simulation. This command is identical to the one in the **Simulation** menu.
- ▣ **Mark** will insert a separate mark into the traces. This function is used to mark the separate information sections obtained during step-by-step operation.

The **Step** and **Release** actions in the API tracer window are the only ones that can restart the system call trace on the next part of the program execution that they trigger, and they must not as such be confused with the functions of the same name that appear in the other ISE graphic contexts. Restarting the simulation using any other button or menu entry will not activate the traces during execution.


10.1.1.6 Triggering Events Manually

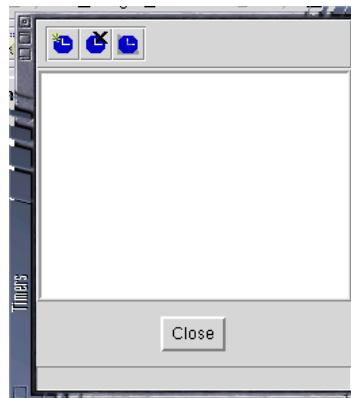
Using the ISE to trigger events manually is controlled by a button for each event defined with a manual generation source. An interrupt event causes an IRQ to be raised on the corresponding node. From the simulation inspector's main window, double-click on the **manual request** icon in our example. The following sub-window will appear:




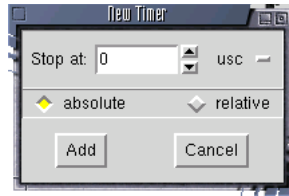
Click on this button to trigger the event. In our example, the **C client_raise_manual** interrupt service routine will be activated, causing a request to be generated by the **Client** node and sent to the **Server** node.

10.1.1.7 Setting Timers





The ISE can automatically hold simulation at certain programmed times. This can be used to break the simulator execution prior to analyzing the various system objects or taking debug actions. To access the timer manager, select the **Timers** command from the **Simulation** menu or choose the  icon from the control bar. The following window is displayed:



- To insert a new time for stopping simulation, click on the  icon. A simulated time selection sub-window is then displayed:




Enter the required time, then validate the operation by choosing the **Add** button. The **absolute** and **relative** selectors are used to specify a time-out calculation from time 0 (**absolute**) or from the simulated current time (**relative**).

- ▣ To inhibit a timer without removing it completely, right click on it and select the **Disable** command from the context related menu. For the opposite result, select the **Enable** command from this same menu to reactivate it. You can also left click on it and choose the  icon from the local toolbar.
- ▣ To toggle the enabled/disabled status of all of the timers using a single action, choose the  icon when no other items are selected from the list.
- ▣ To permanently delete a timer, right click on it, then select the **Remove** command from the context related menu. You can also left click on it and choose the  icon from the local toolbar.
- ▣ To delete all timers using a single action, choose the  icon when no other items are selected from the list. You will be asked to confirm this command.

10.1.2 Using the built-in Debugger

The typical capacities of a graphic debugger have been extended to taking into account the inherently multi-context characteristics (threads, nodes) of a RTOS simulation tool. All of the debugger functions are synchronized with the observation and interaction functions already available through the monitor. This kind of execution therefore offers a mixed simulation control mode using the debugger and the monitor.

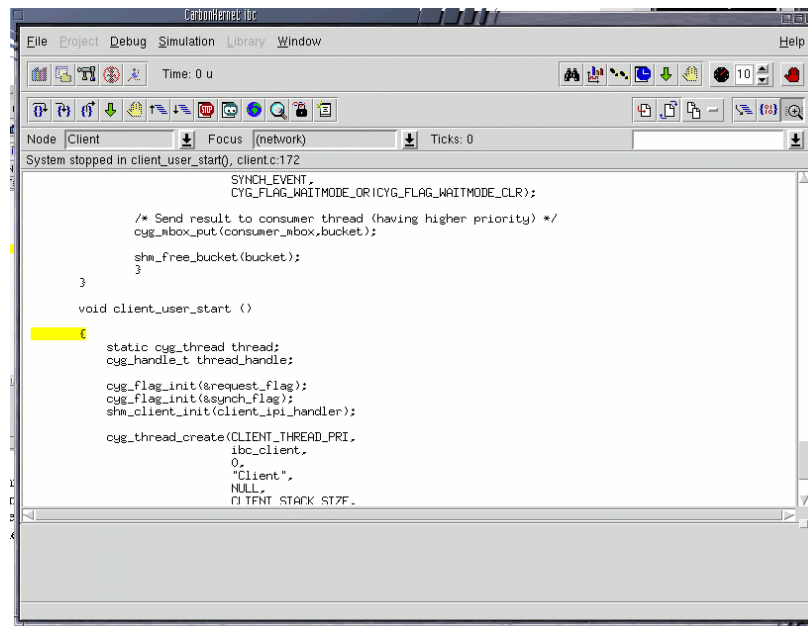
10.1.2.1 Starting the Debugger

The simulator is activated under the control of the debugger using the **Load** command in the **Debug** menu. In addition, the  icon in the toolbar is a direct alias for this command. While the debug engine is loading, a blinking indicator showing the "Loading" condition is displayed in the upper right hand part of the ISE display. At this stage, simulation is not yet active.

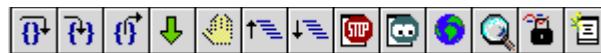
After starting the simulator under the joint control of the debugger and the monitor, a breakpoint is hit at the entry point of the first node, provided it has been instrumented.

The *Run* function handles start-up, without holding up the simulation at time 0, which means that the simulation starts running as soon as it is loaded.

After starting the debugger, our example, **ibc** generates the following display:




In this mode, a toolbar dedicated to debugger functions is displayed on the left of the main window, under the main toolbar. This is enhanced, at the far right in the same plane, by specific source-file management tasks and by a set of operating mode selectors. The general aspect of this new control field is as follows:



The available functions are from left to right, respectively :

- Step-over the current source statement.
- Step-into the current source statement.
- Step-out the current routine.
- Release/Continue simulation.
- Hold/Stop simulation.
- Go one level toward the outer stack frame.
- Go one level down to the inner stack frame.
- Edit code breakpoints.
- Edit data watchpoints.
- Display global variables.
- Examine memory.
- Lock focus on current thread.
- Open a secondary debug frame.

Under the joint control mode, access to system objects via the simulation inspector is gained by choosing the *Inspect* command from the **Simulation** menu or by clicking on the  icon in the monitor control bar.

10.1.2.2 Using Context Focus

The CarbonKernel debugger offers extended control over simulation. The **Node** and **Focus** selectors allow specifying a particular application context to track and visualize.

The **Node** selector lists known nodes. The **Focus** selector lists the threads that are active on the node currently pointed to by the **Node** selector, followed by two generic inputs called **(network)** and **(node)**. The first one represents the global context, regardless of the current node selection. In this case, the last source statement of application code that was executed will be presented, independently of its context. The second input represents all the activity contexts for the selected node, whether synchronous (threads) or asynchronous (callouts, ISR/DSR). In this position, the last source statement executed on the specified node will be presented. Finally, a focus on a specific thread will narrow the application code trace to the code executed by this thread only.


The combined position of the two **Node** and **Focus** selectors determines the scope of some debugger commands:

- ▣ The step-by-step functions (**StepInto**, **StepOut**, **StepOver**) always work on the current context. For example, a step-by-step command placed during an active focus on a given thread will only hold the simulation after this thread has executed a source statement. All of the other parallel activities will continue to run in the background until the next time the simulation is stopped.
- ▣ The conditions assigned to the breakpoint inserted on-line depend on the current focus. A breakpoint can be inserted on-line using the local menu in the source buffer. To access this, right click on the source code buffer, at the statement position you want a breakpoint to be set.


10.1.2.2.1 Changing the Focus

Use **Node** then **Focus** selectors to change the traced context. The ISE automatically updates the source buffer accordingly.

10.1.2.2.2 Asynchronous Code Trace

Use the toggle selector  to validate the asynchronous code trace mode showing source statements executed on behalf of contexts such as kernel callouts or ISR/DSR routines. If this function is not selected, only synchronous thread activity will be traced.

10.1.2.2.3 Multiple Context Display

Choose the  icon to create a new debug frame. There is virtually no limit to the number of simultaneously accessible frames. All of the debug frames are synchronized and have their own focus. This enables obtaining simultaneous views of the status of the various activities observed at a given time.

10.1.2.3 Position in the Source Code and Local Actions

The current position in the displayed source code is illustrated by yellow highlighting or a cursor located in the left hand margin if the *use glyph cursor in source* option is active in the project settings. As CarbonKernel lets the user follow more than one different context at one time, the position shown is always dependent on the context that is displayed according to the active focus selection (see above).

At any time, when simulation is suspended, a local menu that is called up via the right mouse button can be accessed. The actions presented in this menu may vary depending on the current simulation context.

10.1.2.3.1 Setting a Breakpoint

The *Break here* action lets the user add a breakpoint at the source statement located under the mouse pointer. If the focus is not **(network)**, then a sub-menu displays a list of additional conditions for the breakpoint. It therefore becomes possible to make breakpoint conditional on the execution of any activity (thread, ISR/DSR, kernel callout) affecting the node covered by the focus (*if Node "xxx" runs*), or on the execution of a specific thread on this same node (*if Thread "xxx" runs*). Choosing *unconditionally* is the same as validating this breakpoint regardless of the context, including all activities on all nodes. The choice becomes unconditional in relation to the focus. When the focus is **(network)**, the breakpoint is always valid, regardless of the executing context.

10.1.2.3.2 Editing a Breakpoint

If the source statement that is present under the mouse pointer already has a breakpoint, then the *Disable*, *Enable* and *Remove* actions respectively let the user disable, re-enable or definitively remove this breakpoint.

10.1.2.3.3 Running Until the Mouse Pointer

The *Run until* function lets the user place a temporary breakpoint at the source statement under the mouse pointer and then automatically resume the simulation. This breakpoint will be automatically deleted the next time the simulation is suspended by the debugger.

10.1.2.3.4 Searching for a Character String

The *Search string* action allows access to a lexical search function on a character string in the current source file.




10.1.2.3.5 Actions on the Current Selection

If a text selection is active, the character string delimited in this way can be used as the argument for a variable or expression inspection function. The selection is made by double-clicking on any word in the source code. If the expression being looked for is not currently displayed, it is possible to enter it into the freeform text field. Possible actions of the local menu in the source buffer include:

- ▣ *Display* to display the expression value in the local variables window.

- ▣ *Display* * to display the result of de-referencing the expression value in the local variables window.
- ▣ *Type of* to obtain the type of expression in the source code programming language syntax.
- ▣ *Seek* to bring the declaration of the variable or the function proposed as an argument into the source buffer. This action has a shortcut that is called up by simultaneously pressing the **Control** key and the left mouse button, when the pointer is located on the expression.
- ▣ *Find* to look for the next occurrence of the argument in the source text. The find function automatically wraps to the start of the source file when it reaches the end.

10.1.2.4 Using Step-by-Step Mode

- ▣ Choose the  icon from the debug toolbar to resume execution from the current statement and enter the routine (**StepInto** function).
- ▣ Choose the  icon from the debug toolbar to resume execution from the current statement without entering the routine (**StepOver** function).
- ▣ Choose the  icon from the debug toolbar to resume execution from the current statement until exiting the active routine (**StepOut** function).


10.1.2.5 Information on the Current Context

When execution is suspended under the control of the debugger, the function name, file name and source line currently pointed to by the step-by-step mode cursor are displayed over the source text. This information is completed where necessary by the current interrupt masking level (when applied, i.e. > 0), as well as by the specific mode indicators if this is a thread. These modes are as follows:


- ▣ (lock) indicates that the displayed thread cannot be preempted by another thread from the same node (i.e. Scheduler lock).
- ▣ (rrb) indicates that current node is currently undergoing a round-robin scheduling for the priority level the displayed thread belongs to.
- ▣ (asdi) indicates that asynchronous software signals are currently inhibited for the displayed thread.
- ▣ (boost) indicates that the displayed thread temporarily benefits from enhanced priority, as part of the *priority inheritance protocol*.

When these modes are meaningless to the currently simulated RTOS personality, their respective flags never appear.

10.1.2.6 Displaying the Stack


Use the toggle selector  to validate or inhibit the display of the call stack that corresponds to the traced context. Click on a stack level to display the corresponding source code.

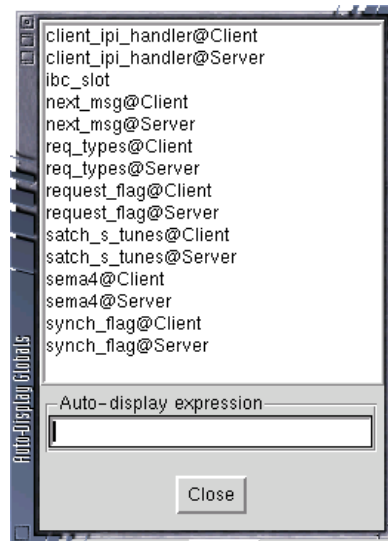
10.1.2.7 Local Variables

Use the toggle selector  to validate or inhibit the display of the local variables for the traced context. Right click on the list of variables to

pulldown the local menu to access the various additional functions. This menu is used in exactly the same way as the one defined for the local menu in the global variables window.

10.1.2.8 Displaying Global Variables

The global variables for the application program are accessed using the  icon in the debug toolbar. Right click on the window displayed, then use the **Select** command in the local menu to choose the variables to display. For our example, the selection sub-window shown below:



Each global variable is instantiated as many times as there are nodes defined in the configuration. When more than one node is active, each global variable is suffixed by the '@' character followed by the name of the instance that it belongs to. This syntax can be reused when manually entering expressions to display using the **Auto-display expression** function (see below).

In our example, two instances exist for each variable; one for the **Client** node, the other for the **Server** node. Double-click on a variable instance to display it. Choose the **Close** button to end the selection. A text entry field left blank and called **Auto-display expression** lets the user type in an expression to display.

Any variable or expression that is selected using this function is automatically recalculated and graphically refreshed after each suspension of execution. All of the expressions supported by the active debugging engine (e.g. GDB) are valid in this context, especially arithmetical expressions. This feature is also accessible from the local variables display window.

In our example, enter the **next_msg@Client** expression, then use **ENTER** to validate and see at all times, the selection index value for messages on the **Client** node.

10.1.2.9 Specifying the Context of a per-node Variable


When a variable has been adapted by the instrumentor to be instantiated for multiple node use in a single simulation, it is possible to specify the reference context that its value will be searched for in a data display expression submitted to the debugger. The principle involves suffixing the variable name with the symbolic node name, separated by the '@' special character.

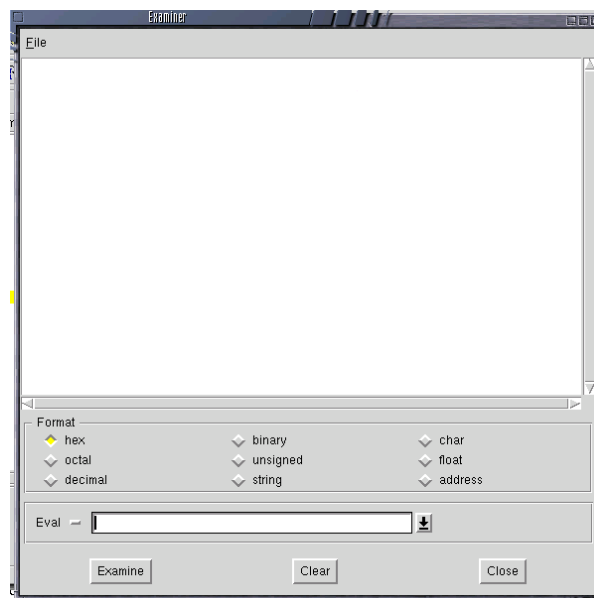
For example, **ibc_handler@Server** indicates the value of the per-node variable **ibc_handler** in the **Server** node context.

If the node name comprises blank spaces and/or characters that may conflict with arithmetic operators or others recognized by the debugger, then the name should be delimited with a set of brackets. For example, **ibc_handler@(A Client Node)**.

Finally, if the node was obtained by cloning, its specification should be followed by its instancing rank in relation to the reference node. For example, **ibc_handler@Server(0)** indicates the first instance of the node created by cloning the **Server** reference node. The above rule relating to delimiting the name also applies in this case, with for example the use of the **ibc_handler@(A Client Node(1))** expression to specify the value of the **ibc_handler** variable in the second clone of the reference node called "A Client Node".

10.1.2.10 Displaying the Memory and Evaluating Expressions

Use the  icon to access the memory examiner. The window below is displayed:



The first frame contains the request results display buffer. The intermediate frame contains a set of format selectors used to check memory display (hexadecimal, decimal, octal and others). The third frame is used to choose the expression to be displayed.

Two modes are available in this context: the raw memory display if the selector in front of the text entry is in the **Dump** position, and the


expression evaluation mode if this same selector is in the **Eval** position. The **Dump** mode displays the content of the memory, from an address expressed in the next text entry, for a number of consecutive memory words using a general format specified by the other selectors. The **Eval** mode will merely evaluate the expression shown in the text field. In this case, the format selector is not operational.

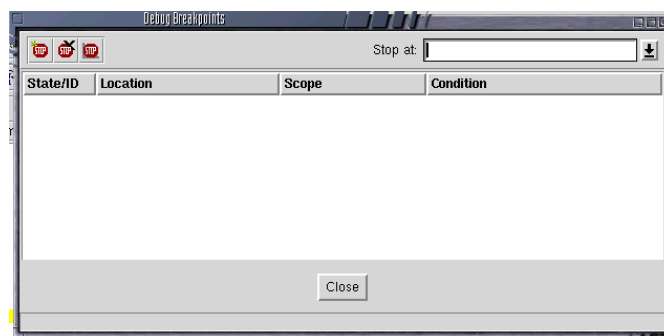
The **Eval** mode also lets the user change the value of the expression displayed by allowing it to be placed after the **=** operator followed by the new value to assign to it. For example, `A= 23` assigns the value 23 to variable A.


When the examiner's window remains open during step-by-step operation, the expression normally displayed in the text entry is re-evaluated each time simulation is stopped.





The **Examine** button forces the evaluation of the current content of the text entry, just like when the **RETURN** key is pressed in the same field. The **Clear** clears the contents of the field.

10.1.2.11 Setting Breakpoints

- Pull down the local menu for the source frame by right clicking on the source statement that should receive the breakpoint. Choose one of the breakpoint conditions proposed by the current focus function. The possible conditions may make the breakpoint dependent on the node or thread currently being presented. One mode allows obtaining an unconditional shutdown regardless of the context (equivalent to the **network** mode). If the breakpoint requires a condition, use the breakpoint editor's **Set condition** function (see below).
- Use the  icon to access the breakpoint manager. In this mode, only unconditional breakpoints may be defined. The manager window is as follows:




- To insert a new breakpoint, enter its location in the text field, then click on the  icon. The syntax used to specify localization depends on the debug engine used. By default, the CarbonKernel ISE uses the GDB engine. For example, state **server.c : 86** to stop the execution at line 86 of the server.c file in our example, then validate this by pressing the **ENTER** key. Or alternatively, state **shm_push_q** to stop the execution at the input to the shm_push_q() function (present in the shm.c file). It is also possible to add a condition to breakpoint triggering use of the **Set condition** function from the local menu.

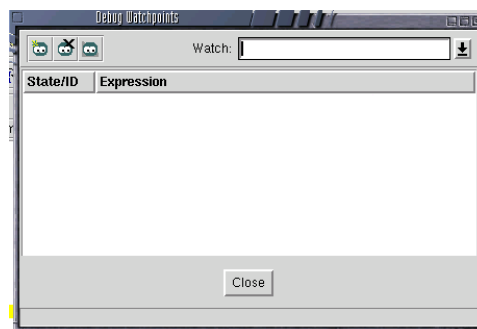
- ▣ To inhibit a breakpoint without removing it definitively, right click on it, then select the **Disable** command from the local menu. To perform the reverse operation, select the **Enable** command from this same menu and reactivate the breakpoint. You can also left click on the breakpoint, and then choose the  icon from the local toolbar.
- ▣ To toggle the enabled/disabled state of all of the breakpoints using a single action, choose the  icon when no other elements are selected from the list.
- ▣ To definitively remove a breakpoint, right click on it, then select the **Remove** command from the local menu. You can also left click on the breakpoint, and then choose the  icon from the local toolbar.
- ▣ To definitively remove all breakpoints using a single action choose the  icon when no other elements are selected from the list. You will be asked to confirm this action.


10.1.2.12 Setting Watchpoints





This function is used to stop the simulation when a variable expression changes value when the program is under control. To use the data watchpoint in a comfortable manner, hardware support is required from the host station, in order to reduce the execution time penalty linked to controlling the change in value to the submitted expression. When a watchpoint is first used, the debugger will request confirmation of the operation to the user if this support proves unefficient at the debug engine level. If the action is confirmed, then a software emulation function eventually provided by the debug engine will be used.

The unavailability of hardware support for controlling memory watchpoints makes their use unreasonable due to the extreme slowing effect that software emulation causes at the debug engine level (e.g. with GDB). In addition, unavailability is most often due to a lack of support from the debug engine itself, rather than to any actual lack of hardware functions at the host processor level.



- ▣ Use the  icon to access the breakpoint manager. From this mode, only unconditional breakpoints can be defined. The manager window is shown below:



- ▣ To insert a new watchpoint, enter the chosen expression in the text entry, then click on the  icon. It is possible to change this expression using the **Edit** function in the local menu.

- ▣ To inhibit a watchpoint without removing it definitively, right click on it, then select the **Disable** command from the local menu. To perform the reverse operation, select the **Enable** command from this same menu and reactivate the watchpoint. You can also left click on the watchpoint, and then choose the  icon from the local toolbar.
- ▣ To toggle the enabled/disabled status of all of the watchpoints using a single action, choose the  icon when no other elements are selected from the list.
- ▣ To permanently remove a watchpoint, right click on it, then select the **Remove** command from the local menu. You can also left click on the watchpoint, and then choose the  icon from the local toolbar.
- ▣ To definitively remove all watchpoints using a single action choose the  icon when no other elements are selected from the list. You will be asked to confirm this action.

10.1.2.13 Controlling Simulation


- ▣ Simulation is automatically held at time "0", after general debug initialization. Select the **Release** command from the **Debug** menu to start execution. This command has a double alias in the form of the  icon present in the debugger control bar and in the monitor that is displayed on start up. You can use this command to resume simulation after any kind of break state. To stop the simulation again and return control of it to the debugger, select the **Hold** command from the **Debug** menu, or click on the  icon in the control bar.
- ▣ To definitively stop the debugging and simulation session, select the **Kill** command from the **Debug** menu.
- ▣ To restart simulation, with or without the debugger, use the **Restart** command from the **Debug** or **Simulation** menus as appropriate.

The simulation control actions accessible via the monitor are also available during a debug session using the debugger.


10.1.2.14 Selecting the Source File

The following actions directly affect the contents of the source buffer displayed by the debugger.


10.1.2.14.1 Back to the Current Instruction

Click on the  icon to display the current breakpoint, in the source buffer. This function is useful for returning to the current instruction after browsing between a number of source files.

10.1.2.14.2 Reloading the Current File

Click on the  icon to force the system to re-read the file currently in the source buffer.

10.1.2.14.3 Browsing Recent Files

The graphic selector  is used to access the list of the last eight files presented in the source buffer. The other less recent files are also retained and can be accessed via the selector's *More...* input when this is available. In this case, a selection list lets the user browse all already open files.


10.1.2.14.4 Loading a New Source file

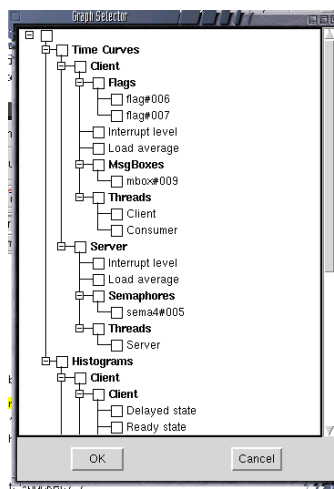
Choose the **Open...** input from the **File** menu in the main menu bar in order to access a file selector. After validating the action, the source code contained in this file will be presented in the current buffer.


11. Displaying Statistics Graphs

Some system objects created by the simulation models export one (or more) displays of their current status to a dedicated plotter, accessible from the ISE monitor. For example, real-time thread transitions are displayed in state diagram form, presenting every possible state (suspension, delay, active, etc.) over time. Some models may also propose a graphic illustration of simulated instantaneous processor workload using a curve or a set of histograms showing the distribution of thread states during the simulation. The graphs proposed for display are therefore dependent on the characteristics of the simulation models used.

11.1 Selecting Graphs

To access the plotter functions, select the **Plotter** command from the **Simulation** menu or click on the  icon in the monitor toolbar after starting simulation. The graph selection screen is displayed:



The selection screen only appears when no graphs are currently displayed, otherwise access to the graph trace window is gained directly. At any time you can return to the selector using the **Select** command from the **File** menu in the plotter window or by using the  icon.

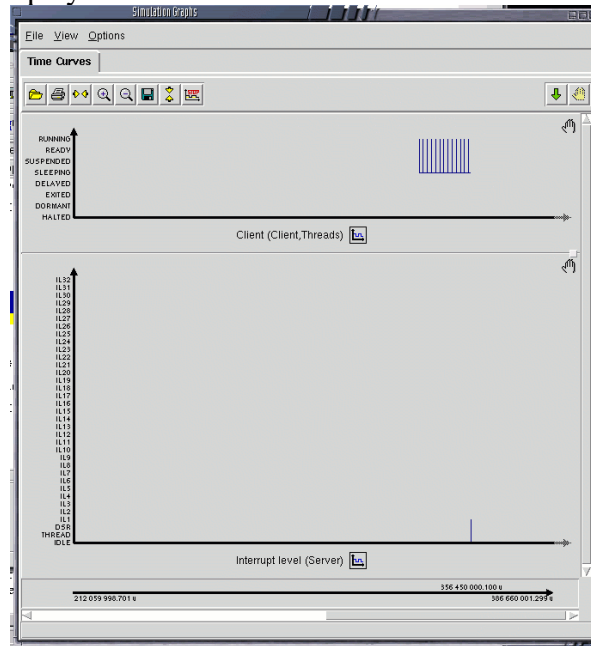
As you can see in the previous view, the graphs are listed in hierarchical order, by graphic type first (**Time Curves** for the time curves, **Compounds** for the composite curves and **Histograms** for histograms), then by node assignment and finally by type of simulation objects.

All of the curves that exist at this point in the simulation are available on screen. The selection state for each node is carried via the lower branches to the graphic objects themselves are reached. The selected curves are initially displayed in the order of their appearance in the list.

Simulation can export new curves throughout its operation, depending on the performance of the models involved. For example, creating a thread generally causes the appearance of a new state diagram that represents it. In this case, the graph selector will be enriched in real-time.

In our example, choose the **Client** and **Interrupt level** curves from the **Client** and **Server** nodes respectively, that show the state of the client thread and the

current server node's interrupt level. After validating this function using **OK**, the plotter window is displayed:



11.2 Display by Type of Graph

11.2.1 Time Curves

Curves indexed on the simulation time are grouped so that they can be compared. The time value is assigned to the abscissa, with a sliding left limit based on its evolution or the current presentation choices. We will use **xMin**, **xCur** and **xMax** respectively as the minimum, current (i.e. simulation time reached prior to the last suspension) and maximum values for this axis in the rest of this document. Click on the **Time Curves** tab in the curve tracer to display this kind of curve.

11.2.2 Composite Curves


A number of time curves of the same type can be assigned together to the same composite curve. They retain their separate properties but appear grouped when displayed in the same graphic field.

11.2.3 Histograms

The histograms are grouped under the **Histograms** tab in the tracer. The **Y_axis** is used to illustrate the values reached by the different distributions displayed.

11.3 Controlling the Simulation

11.3.1 Continuing/Stopping Simulation

The simulation in progress can be stopped temporarily and restarted from the plotter, using the  and  icons respectively in the control bar. These two

functions are identical to the ones that can be found in the debugger and built-in monitor interfaces.

11.3.2 Setting Breakpoints

Breakpoints can be set on the time curves shown. The transition to one of the states chosen from a state diagram, or reaching a numerical threshold in a time graph can therefore trigger an automatic stop to simulation. Control is then returned to the ISE, which then synchronizes all of the views available on the breakpoint (updating the status of inspector objects, the code and the variables displayed by the debugger etc.).

To access this function, right click on the title of a time curve to call up the local control menu for this curve, pulldown the **Mode** sub-menu, then choose **Breakpoint**.

11.3.2.1 Breakpoints in a State diagram


Once the **Breakpoint** function is validated, a list of possible diagram states is displayed in a selection sub-window. Check the selector for each status required. The **Close** button ends the selection process.

11.3.2.2 Breakpoints in a Time Graph

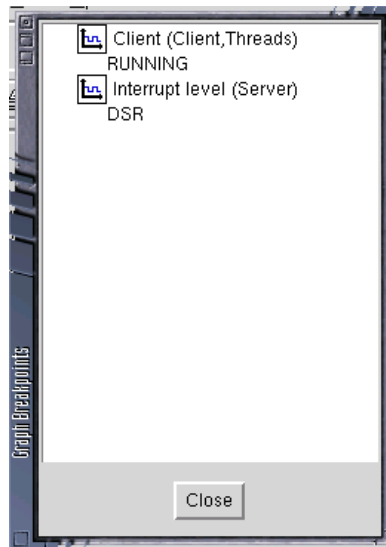
Once the **Breakpoint** function is validated, a sub-window is displayed for entering the threshold value. Enter this value, which will stop the simulation when it is reached. The **Close** button creates the breakpoint for the current threshold value. The **Remove** button deletes the breakpoint for the displayed value.

An accelerated mode for setting breakpoints can be accessed by directly double-clicking on the chosen part of the curve, representing the ordinate value for the status or threshold.

11.3.3 Displaying and Checking Breakpoints

A function lets the user gain a summary overview of all of the breakpoints set on the time curves. This function will also allow disabling or removing these breakpoints, either individually or curve by curve. Use the **Breakpoints** option from the **View** menu or click on the  icon to access this function.

A window like the one below is displayed:




The breakpoints displayed are grouped by the curve they belong to. Right click on a breakpoint value to pulldown the local control sub-menu. The **Remove**, **Disable**, and **Enable** choices respectively will remove, disable or re-enable the selected breakpoints. If you would like to apply these actions globally to all of the breakpoints on a given curve, right click on the name of the curve as displayed in this same window and use the required sub-function.

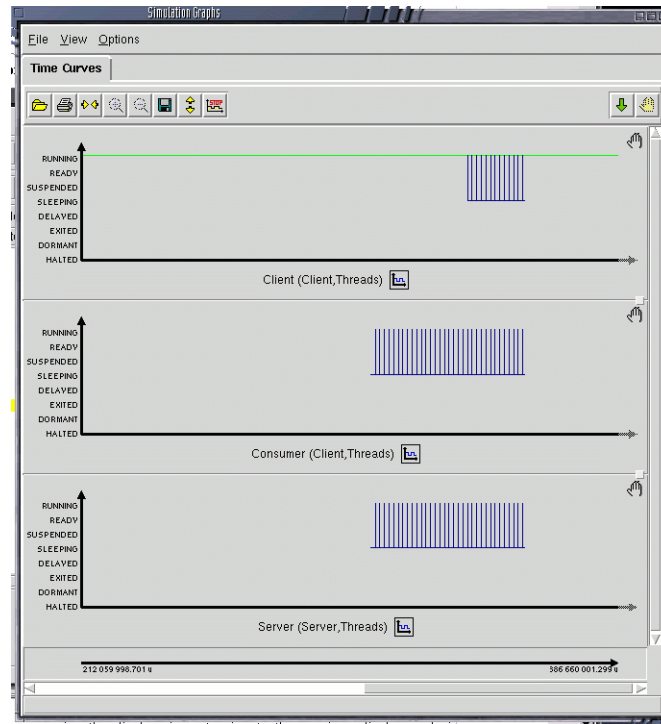
The **Close** button closes the breakpoints control sub-window.


11.4 Scale Compression

11.4.1 Y Compression


When the initial height of the curves does not allow their complete display and requires the use of the vertical scroll bars, then it is sometimes useful to vertically compress these objects. Use the **Y-compress** function in the **View** menu or the  icon to reduce the height of these objects so that they can be displayed on a single graphic page. Note that this option is automatically disabled if too many objects are already compressed for display. This option is available for time curves and histograms.


In our example, vertically compressing all the thread states curves would generate the following display:




Vertically uncompressing the display, i.e. returning to the previous display scale is possible using the **Y-uncompress** function from the **View** menu or by using the  icon.

11.4.2 X Compression


It is often convenient to be able to display what a curve looks like over the entire simulation time range. Use the **X-compress** function from the **View** menu or the  icon to bring all of the stored points from all of the time curves back to a single graphic page. This operation is performed by applying an abscissa scaling function that changes the displayed time/pixel ratio. The left and right limits of the curve are brought back to 0 and xCur respectively.

Horizontally uncompressing the display, i.e. returning to the previous display scale is possible using the **X-uncompress** function from the **View** menu or by using the  icon.

11.4.3 Zoom In

The zoom in function expands the time/pixel scale by 200%. The left hand limit (**xMin**) is retained as closely as possible, while the display is recalculated for all of the time curves displayed. This function is a gradation of the X compression function described previously. It is called up by choosing **Zoom In 200%** from the **View** menu or using the  icon.


11.4.4 Zoom Out

The zoom out function contracts the time/pixel scale by 50%. The right hand limit (**xMax**) is changed to match the new scale, while the display is recalculated for all of the time curves displayed. This function is symmetrical to the zoom in function described previously, but is however applied using a lower scaling factor. It is called up by choosing **Zoom Out 50%** from the **View** menu or using the  icon.

11.5 Composite Curves

In order to obtain a single curve made up from the states of a number of other compatible curves (i.e. curves of the same type and in the case of state diagrams, with the same states), it is possible to use the "drag & drop" method based on the time curves title bar. To do this, simply left click on the title bar of one of the curves to group, then drag the mouse to the title bar of the other curve to group, whether it is a composite curve or not, then release the left mouse button. In the former case, the first curve will be added to the second composite, in the latter case, the two curves will form a new composite. In this latter case only, the plotter will prompt the user for the name of the newly created composite curve.

11.6 Placing Graphs

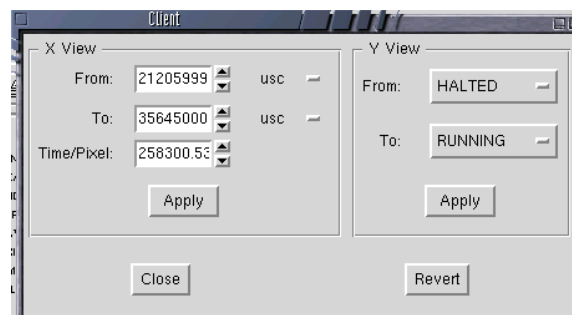
Graphs may be arranged for display using a simple placing mechanism, triggered by a "drag & drop" operation performed on the  grab icon provided on each one. The graph located at the source of the move action will be inserted before the one located at the move destination.

11.7 Selection and Cross Hairs

11.7.1 Actions on Graph Sections

Some graph analysis actions require expanding or compressing a specific part of the time curve trace. The graphic selection concept is used to select a precise area of application for a change of scale. To call up this function, right click on the title of a time curve to call up the local control menu for this curve, pull down the **Mode** sub-menu, then choose **Selection**.

A sub-window like the one shown below will then be displayed:



The **X-View** heading in this window lets the user control the time/pixel ratio (i.e. the abscissa) over the current selection. The **Y-View** heading is used to control the field shown by the ordinates.

In parallel with the window display, a selection rectangle is shown on the selected curve. Bring this rectangle over the graphic portion that you wish to assign to the selection. To do this, left click on the point that represents the top left corner of the rectangle, then drag the pointer to the lower right corner, while holding the mouse button down. The selection made is determined by the rectangle formed when the left mouse button is released. You can rework an existing selection by holding down the **CONTROL** key while performing the operation using the left mouse button.

During this operation, the current limits of your selection are displayed in real-time in the control window, showing both the abscissa and ordinate values. You can change these limits manually using the data entry fields provided.

In the **X-View** heading, **From** and **To** respectively control the **xMin** and **xMax** values displayed. **Time/Pixel** is the display ratio between a graphic pixel and its time correspondence.

In the **Y-View** heading, the system will display the limits-states for a state diagram, or the minimum and maximum values in ordinates, for a time graph. In the former case, only those states present between the two limits (inclusive) will be displayed. In the second case, only those points between the minimum value and the maximum value (inclusive) will be displayed.

Apply the new display parameters by clicking on the **Apply** button. To revert to the previous state, use the **Revert** button.

There is a quick way to access the selection function, by simultaneously pressing the left mouse button and the **SHIFT** key while directly selecting the target curve. Once the selection has been made, right click on the chosen curve, then validate **X-Apply** or **Y-Apply** respectively to change the scale or the corresponding limits.

11.7.2 Using the Cross Hairs

Left click on a point on a time curve or a histogram to see its coordinates. The coordinates will be displayed at the top of the selected curve. The cross hairs let you move along this curve while holding down the left mouse button, for a real-time display of the corresponding coordinates.

11.8 Other Local Functions

The graphs shown all have a local menu that lets the user access a set of individual functions. To access this menu, right click on the title of the graph.

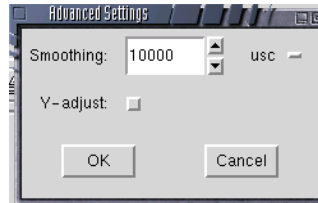
11.8.1 Local Time Curve Functions

11.8.1.1 Seeking the Next Point

The **Seek** sub-function in a curve's local menu lets the user seek the next or the previous point received from the simulation by looking forward (**Forward** input) or backward (**Backward** input) within the current view. This function is useful when looking for the next or the previous transition in a state diagram. The minimum limit **xMin** is moved to bring the point found onto the display. If no point is found, an audible beep will sound, indicating that the seek action has failed to find anything.

11.8.1.2 Advanced Settings

Simple time graphs (i.e. time curves excluding state diagrams) have an additional setting feature that can be accessed from the **Advanced** sub-function in their local menu. The advanced settings sub-window looks like this:



The **Smoothing** parameter is a smoothing constant applied to the graph along its horizontal scale. It is expressed as a simulated time value.

Selecting the **Y-adjust** parameter will automatically adjust the maximum ordinates limit according to the points received from the simulator. If not, the limit will not be readjusted to match the maximum value received from the simulator.

Choose **OK** to validate the changes made to the settings, or **Cancel** to cancel the operation.

11.8.2 Local Histogram Functions

11.8.2.1 Display Mode

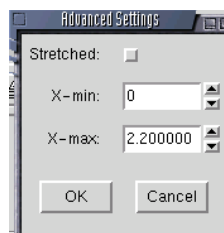
The histogram display mode can be controlled by the **Display** sub-function. The **Density** option specifies the probability density, the **Repartition** option specifies the repartition function. By default, the probability density is applied.

11.8.2.2 Representation Type

The histogram representation type can be controlled by the **View** sub-function. The **Absolute** option specifies the absolute mode, the **Relative** option specifies the relative mode. By default, the relative mode is applied.

11.8.2.3 Advanced Settings


All histograms have a additional settings that are accessed by the **Advanced** sub-function from the local menu. The advanced settings sub-window looks like this:



Selecting the **Stretched** parameter will extend the histogram trace surface to cover all of the drawing surface available along the horizontal axis. Otherwise, a default horizontal size is given to the graph, which may vary depending on the distribution presented.

X-min and **X-max** respectively are the simulated time minimum and maximum values used to represent the distribution.

11.8.2.4 Updating

Unlike time curves, histograms are not updated in real-time during simulation. Updating should be triggered on demand using the **Update** function in the **View** menu or using the  icon.

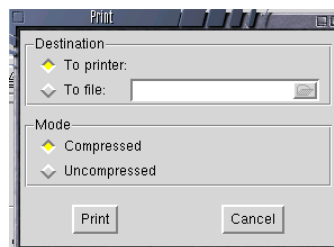
11.8.3 Common Functions

11.8.3.1 Changing Colors

The color used to draw the graphs may be changed using the selector that is accessed via the **Color** sub-function in the local menu for these graphs.


11.8.3.2 PostScript Printing

Time curves and histograms can be printed out in PostScript format, either to a file or directly to the chosen printing peripheral. From the graph's local menu, use the **Print** sub-function to access the Print sub-window. The following window is displayed:




Choose the destination, **To printer** for a direct print out to a printer, **To file** to route the print out to a file. In the latter case, the **filename** text entry next to the selector must be filled-in. For a direct print out, the current **Print command** settings in the simulation configuration parameters are used. These settings are accessible from the **Tools** page in the **Settings** window in the ISE **Project** menu.

Horizontal curve compression is applied by default, so as to present the entire simulated time range in the PostScript trace. You may disable this option by choosing the **Uncompressed** mode.

A global print command for all of the graphs present on the current page (time curves or histograms) can be called up from the **Print** command in the **File** menu or using the  icon.

11.8.3.3 Removing the Graph

The **Remove** sub-function, accessed from the local menu for a graph lets the user remove it from the display, regardless of its type. It may be restored using the graph selector accessed via the **Select** command in the **File** menu or via the  icon.

11.9 General Options

The plotter offers a number of options that affect its overall performance. These options are accessible via the **Options** menu.

11.9.1 Adjusting Abscissas

The **X-adjust** mode is used to choose automatic readjustment of the **xMin** and **xMax** limits when plotting the time curves, depending on the minimum and maximum values received from the simulator. This allows retaining only one active graphic page at any time during the simulation, and therefore avoids the need to use the horizontal scroll bar. This mode is disabled by default.


11.9.2 Scroll Lock

The **Scroll lock** mode indexes time curve horizontal scrolling to mouse motion, when using the horizontal scroll bar. Otherwise, the display is only refreshed when the mouse button is released. This mode, when it is enabled, requires a high trace speed from the host station, due to the numerous display refreshes that take place. This mode is disabled by default.

11.9.3 Auto-Select Color

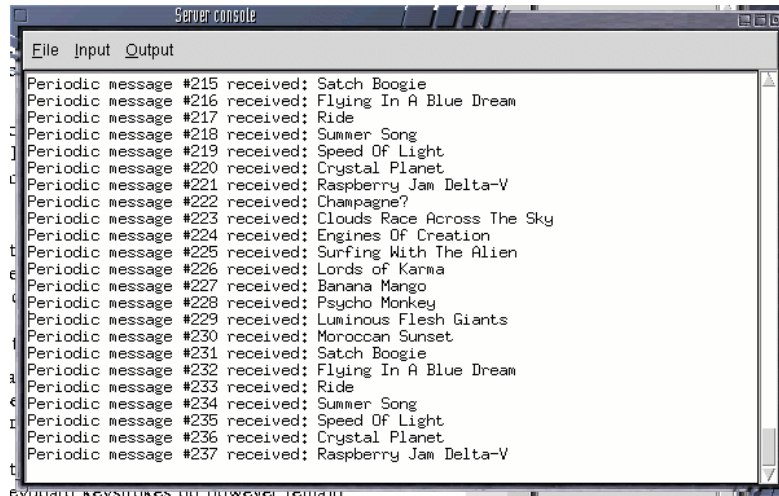
The **Auto-select color** mode lets the tracer choose the colors to assign to new curves. This mode is initialized by default.

11.9.4 Auto-Save Session

All of the current plotter session parameters can be saved on demand using the **Save** command from the **File** menu or using the  icon. This includes the list of curves displayed, the composite curves that are active, the position of the breakpoints on the different curves, the advanced configuration settings, etc. The **Auto-save session** option lets the user choose to automatically run this command at the end of each session, when the simulator is shut down.

12. Using the Terminal Console

The VRTOS create a system console channel for each node. A graphic application representing an ASCII terminal is then connected to each channel.



The standard simulator standard input and output streams (i.e. stdin/stdout/stderr) are automatically directed to this terminal, just like they could be in a real situation to the target console channel.

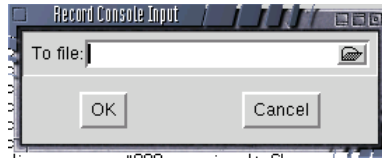
It is possible to inhibit the redirection of simulator data flows using a command line execution mode on the simulator via the **-Xn** option. Refer to the **Command Lines** section that covers the simulator, for further information on this heading.

The CarbonKernel terminal has a dual-function:

- On the one hand it allows interacting with the simulated RTOS using its console channel; the outputs generated by the user code during execution will be logged in the assigned display window and in the same way characters entered from the keyboard in the context of normal terminal operation will be immediately forwarded to the node they are intended for. Outputs can be exported to an ASCII file.
- On the other hand it provides the ability to automatically replay keystrokes, to and from an ASCII file. During recording, all of the characters entered are saved in the file along with the simulated time when they were received by the VRTOS. During replay, the characters are extracted from the file, then re-injected into the simulator, either at exactly the same simulated time or immediately, depending on the selected replay option. During input file replay, interactive keyboard keystrokes do however remain valid and are sent to the simulator.

12.1 Recording an Interactive Session

Use the **Start recording** command in the **Input** menu in the terminal window to trigger the keystroke record mode. A sub-window used to specify the access path to the recording file is then displayed:

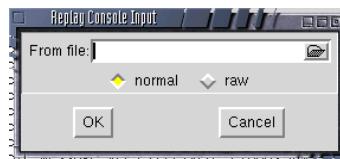


After entry, use the **ENTER** key to validate or click on the **OK** button.

During this mode, a red sphere is displayed intermittently in the top right of the terminal display. This mode ends when the **Stop recording** command from the **Input** menu is validated.

12.2 Replaying an Interactive Session

Use the **Start replay** command from the **Input** menu in the terminal window to trigger the keyboard keystroke replay mode. A sub-window that lets the user enter the replay access path and the replay mode is then displayed:



The mode selector offers a choice between two types of replay:

- ▣ Choose **normal** so that the characters contained in the file will supply the simulator input at the exact same simulated time as they were previously recorded in the file.
- ▣ Choose **raw** if you would like the characters to be injected immediately, just like if they had been received immediately. This mode is perfectly suitable for replaying an interactive command sequence when matching the simulated timeframe is unnecessary.

After entry, validate the selection using the **ENTER** key or click on the **OK** button.

During this mode, a green arrow is displayed intermittently in the top right of the terminal display. This mode ends manually when the **Stop replay** command from the **Input** menu is validated, or automatically if all of the file characters have been replayed by the simulator.

12.3 Manually Creating a Replay File

Here is the description of the file format expected by the simulator as an input for the two types of replay modes described previously. The files obtained in this way can be used as source files for the **Start replay** function.

12.3.1 Format for Time Based Replay ("normal" mode)

- ▣ The first line must start with the "#\$@timelog" marker for the rest of the file to be correctly interpreted. Each subsequent file line starts with the simulated time of data input (time stamps must be absolute and increasing) including its time unit (usc, msc, sec), ending with the ':' separator followed by the character to input, or its ASCII code in octal format preceded by a \ character (e.g. \012 for LF). C language meta-characters

are also accepted (e.g. `\r\b\f\n...`). A line starting with a '#' character is considered to be a comment. Blank lines are allowed and ignored. For example, the following file fragment provides the insertion sequence for the word "CarbonKernel" on the console's input channel, with its time values expressed in micro-seconds:

```
# $@timelog
186 usc:C
186.1 usc:a
186.92 usc:r
190.5 usc:b
192.301 usc:o
192.302 usc:n
192.700 usc:K
197.2 usc:e
198.78 usc:r
198.79 usc:n
200.1 usc:e
202.8 usc:l
```

12.3.2 Format for Immediate Replay ("raw" mode)

In this mode, the file may either be compatible with the time based format described above, or contain a succession of characters that will be sent without modification. If the file contains time data, it will simply be ignored when it is replayed and only those characters located after the ':' delimiter will be sent without delay to the console's input channel. On the contrary, if the file does not have a "# \$@timelog" header, the characters that it contains will be sent as a block, including the line feeds as significant characters. In this format, no comment field is recognized. Sending the word "CarbonKernel" on the console's input channel is the same as simply storing this character string in a raw ASCII format file:

```
CarbonKernel
```

12.4 Saving Terminal Outputs

Use the **Save** command from the **Output** menu in the terminal window to save the contents of the output buffer in an external ASCII file. The filename entry procedure is the same as the one used for previous commands.

12.5 Deleting the Terminal During Simulation

Use the **Quit** command from the **File** menu in the terminal window to force an end to the terminal application. The console channel for the assigned node will be closed automatically by the simulator at the same time.

This command must be used with care as it may cause the failure of the simulated services that are dependent on the console channel.

13. Command Lines

13.1 ISE Start Options

The CarbonKernel ISE has the following set of start options available:

- ▣ **-f <project-file>**, automatically loads the simulation project designated by its access path when the ISE is initialized. This option will override the default behaviour to pre-load the last active project.
- ▣ **-R <repository>**, gives the access path to the ISE's default resources repository. This value is only used when the ISE is first started in a user environment to build the initial database. This option will override the default value set to **<private-dir>/ck.rr**, where **<private-dir>** is the name of the directory created by the ISE during the very first user session. It is located in the root of the user's account and is called **.ck/<arch>** where *arch* is the identifier for the current platform architecture.
- ▣ **-g**, forces simulation to start in debugger mode, as soon as the ISE initialization is finished. Combined with the **--f** option, this selection allows the automatic execution of any simulation project. The **--g** option is mutually exclusive with options **-x** and **-p**.
- ▣ **-x**, forces simulation to start in monitor mode, as soon as the ISE initialization is finished. Combined with the **--f** option, this selection allows the automatic execution of any simulation project. The **--x** option is mutually exclusive with options **--g** and **-p**.
- ▣ **-p**, is an internal option used by the simulator to run a slave monitor interface when simulation is started from the command line.
- ▣ **-q**, unconditionally shuts down the ISE as soon as simulator execution is finished. By default, the ISE remains active after the end of a simulation, except in the case of the slave mode designated by the **-p** option.
- ▣ **-u**, inhibits reloading of the last ISE's working context, especially its last open project. By default, the ISE restores the last project, along with some general display characteristics like the geometry of the graphic window that it uses.
- ▣ **-v**, sends the ISE version number to the standard output.
- ▣ **-e <export-file>**, used in combination with the **-f** option allows exporting the content of the specified project in ASCII format along with the resources repository contents attached to this project. In other words, the ASCII file obtained will contain all of the information required to rebuild an identical CarbonKernel working environment. This rebuild action can be performed using the **-f** option in the ISE command line or by using the **Library** menu input function provided in the same tool. The ISE will automatically end its execution after exporting.
- ▣ **-i <import-file>**, used in combination with the **-f** option allows importing the content of the specified project in ASCII format along with the resources repository contents found in the ASCII file. This operation is symmetrical to the export function obtained using the **-e** option. If the project specified using the **-f** option does not exist, it is automatically created. If not, its contents are replaced by the components found in the imported file. The

resources repository is incrementally updated with the missing or modified elements found in the imported file. The ISE will automatically end its execution after importing.

13.2 Simulator Start Options

Any simulator generated using the standard procedure described in chapter 3 includes the following set of start options:

- ▣ **-C <config>**, specifies the name of the simulated configuration to run. This option is ignored if **--f** (or **--F**) is used. In the latter case, the active configuration name is taken from the project file stated as the argument.
- ▣ **-R <repository>**, specifies the access path to the resources repository from which the simulated configuration designated by the **-C** option will be taken. This option is ignored if **--f** (or **--F**) is used. In the latter case, the active repository is taken from the project file stated as the argument. If no option sets the access path to the resources repository, the **<private_dir>/ck.rr** file will be used, where **<private_dir>** is the directory name created by the ISE when the very first user session is run. It is located in the root of the user's account and is called **.ck/<arch>** where *arch* is the identifier for the current system architecture.
- ▣ **-F <profile>**, initializes the simulator using configuration parameters found in the project file stated as the argument, but without attaching the ISE to the simulation. This mode enables the matching of the resources repository and simulated configuration to be used, ready for running without a graphic interface.
- ▣ **-f <profile>**, triggers running the ISE in "slave" mode, allowing graphic interaction with the simulator. Only the CarbonKernel monitor functions are available in this mode (i.e. No debug). The argument entered is the access path to the project file that must be pre-loaded by the ISE. This file will also provide the information on the resources repository and the simulated configuration to be used.
- ▣ **-X** is a sub-option introducer used for setting the simulator's boolean parameters. This option must be followed by the list of flags to be activated. For example: **-Xhn** inhibits the creation of node consoles while retaining the standard simulator's standard streams on the original channels assigned by the host system. Valid flags include:
 - ▣ **«n»**, inhibits the redirection of standard simulator's streams to the console terminal assigned to each node. By default, the standard streams are redirected to/from the terminal assigned to the active node.
 - ▣ **«h»**, eliminates automatic console terminal creation from the nodes when simulation is started. This option implicitly eliminates standard streams redirection (refer to the **«n»** option), except if the simulator is controlled by the debugger built-into the ISE. In the latter case, the standard streams are redirected to the null channel (refer to the **«z»** option).
 - ▣ **«z»** redirects the standard streams to the null channel. This option has the effect of inhibiting character acquisition on the standard input, and discards standard and error outputs.

- ▣ **«w»**, causes automatic simulation suspension on receiving alert messages. This option is only active during interactive simulation, in the presence of the graphic monitor. By default, a warning does not suspend the simulation.
- ▣ **«a»**, causes automatic simulation suspension on receiving a trace with the alert attribute set. This option is only active during interactive simulation, in the presence of the graphic monitor. By default, a trace message does not suspend the simulation.
- ▣ **«l»**, eliminates checks on code sections run in time-locked mode. By default, the simulator generates an alert if such a section exceeds 10,000 source statements, considering that a potential error linked to the absence of a section close may have occurred.
- ▣ **«c»** redirects the transmissions made to the console terminal from the CKPI services used (i.e. `ckPutChar()`, `ckPutString()` etc. to the standard output stream. The effect of this flag is to allow the display of outputs caused by these services if the console terminals are missing.
- ▣ **«m»** activates a conservative, but slower, management mode for the multi-tasking kernel that is internal to CarbonKernel. Running this mode is generally imposed for any use of the simulator in conjunction with a memory profiler that works by instrumenting the application's binary code.
- ▣ **«g»** inhibits the check performed on the maximum number of alert messages tolerated prior to aborting the simulation. By default, the simulation is automatically abandoned after receiving more than 100 alert messages.
- ▣ **«b»** and **«t»** are flags for internal use that are not accessible to the user.
- ▣ **-t <time[s,m,u]>**, specifies the simulation time limit (simulated time). By default, the simulation duration is infinite. This option is equivalent to the ISE's **Simulation time** parameter that applies to the project.
- ▣ **-w <time[s,m,u]>**, specifies the simulation warm up period before any statistical measurements are made. This option is equivalent to the ISE's **Warmup time** parameter that applies to the project.
- ▣ **-s <nsamples>**, states the number of statistical samples that will be collected by the simulator during the simulation duration. If this parameter is null, then a single sample will be collected at the end of the simulation. If the simulation duration is infinite, no sampling will be performed. In all cases, the statistical samples will be taken periodically at a frequency that equals the simulation duration divided by the number of samples required. By default, only one sample will be taken.
- ▣ **-d <directory>**, changes the simulator's current directory to the one specified in the parameter prior to execution. By default, the current directory is retained.
- ▣ **-l <errlogfile>**, specifies the access path to the file used to store the warning messages generated by the simulator. By default, the messages are sent on the simulator's standard error output.
- ▣ **-v**, used along in the command line, this option sends the version number of the CarbonKernel's base system on the standard output. When used with

start options, this command will cause the display of type and version information for the different simulation models that are instanced.

- ▣ **-z <speed>**, controls the host simulation speed using a factor that runs from 1 to 10. 1 corresponds to the maximum slowing factor.
- ▣ **-b, -p, -u, -k** are reserved internal options.
- ▣ **-Q** is an options prefix that is reserved for the user code. All of the options prefixed in this way may be retrieved unchanged using a service included in the CKPI interface and will be located in the local arguments vector. For example, the occurrence of the **-Qz <file> -Qf** parameters in the command line will be transcribed to the local arguments vector in the form of **-z <file> -f**. Note that there is in this case, no reason not to reuse option keys taken from the simulator command line, provided they are masked by the **-Q** prefix.

Index

- A
 - Add-ins 6, 13, 20, 25p.
 - architecture 8, 18pp, 24, 68p.
 - autoload21
- B
 - Breakpoints 52, 57p.
 - BSP 12
- C
 - ckcc 9pp, 13pp
 - CKPI 9, 37, 70
 - composite 56p., 60, 64
 - Compression 59
 - configuration 10pp, 19pp, 27pp, 31, 33, 35p., 51, 63p., 69
- D
 - DSR 33, 44, 48p.
- E
 - eCos 5, 10, 12, 15, 17, 20pp, 29, 41, 43
 - event-driven 5, 8, 10
 - Export 39
- F
 - Focus 38, 44, 48
 - FROGS 5p., 8p., 12p., 20p., 25, 35
- G
 - gcc 10, 14p.
- H
 - Histogram 56p., 62
- I
 - Import 39
 - inspector 34, 38, 42p., 45, 47, 58
 - Instrumenter
- C/C++ 10
 - IRQ 33, 45
 - ISR 33, 44, 48p.
- M
 - Magnets29
 - main() 17
 - modules 15p., 21, 29
- N
 - netshared 11, 16p.
 - net-shared 13pp
 - nid 11, 25
- P
 - performance 6, 10, 56, 64
 - personality 5, 8p., 12p., 50
 - PostScript 63
 - priority inheritance protocol 50
- R
 - recording 65p.
 - replay 65pp
 - round-robin 50
 - RTC 10
- S
 - SDD 9
 - SDDK 9, 12
 - Selection 44, 49, 60
 - selftest 7
 - SIMEX 8, 10
 - Smoothing 62
 - Stack 50
 - state diagram 56pp, 60pp
 - static constructors 12
- T
 - Target Warp 10, 41
 - terminal 65pp, 69p.

time charge 5, 14, 16
time-locked 14p., 70
Timers 45
V
virtual RTOS 5, 8pp
VRTOS 8, 13, 33, 65
Z
Zoom 60