

AFRL-IF-WP-TR-2001-1553

**TIMEBENCH A VISUAL ENVIRONMENT FOR
THE DESIGN AND INTEGRATION OF OBJECT
ORIENTED REAL-TIME SYSTEMS**



**TIMESYS CORPORATION
4516 HENRY STREET
PITTSBURGH, PA 15213**

SEPTEMBER 2001

FINAL REPORT FOR PERIOD 29 APRIL 1999 – 29 SEPTEMBER 2001

THIS IS A SMALL BUSINESS INNOVATION RESEARCH (SBIR) PHASE II REPORT

Approved for public release; distribution unlimited

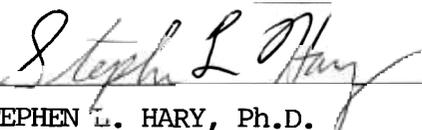
**INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334**

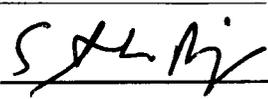
NOTICE

USING GOVERNMENT DRAWINGS, SPECIFICATIONS, OR OTHER DATA INCLUDED IN THIS DOCUMENT FOR ANY PURPOSE OTHER THAN GOVERNMENT PROCUREMENT DOES NOT IN ANY WAY OBLIGATE THE US GOVERNMENT. THE FACT THAT THE GOVERNMENT FORMULATED OR SUPPLIED THE DRAWINGS, SPECIFICATIONS, OR OTHER DATA DOES NOT LICENSE THE HOLDER OR ANY OTHER PERSON OR CORPORATION; OR CONVEY ANY RIGHTS OR PERMISSION TO MANUFACTURE, USE, OR SELL ANY PATENTED INVENTION THAT MAY RELATE TO THEM.

THIS REPORT IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS, IT WILL BE AVAILABLE TO THE GENERAL PUBLIC, INCLUDING FOREIGN NATIONS.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.


STEPHEN L. HARY, Ph.D.
Project Engineer
Advanced Architecture &
Integration Branch


STEPHEN L. BENNING
Team Leader
Advanced Architecture &
Integration Branch


DAVID A. ZANN, Chief
Advanced Architecture & Integration Branch
Information Systems Division

Do not return copies of this report unless contractual obligations or notice on a specific document requires its return.

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> <i>OMB No. 074-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE SEPTEMBER 2001	3. REPORT TYPE AND DATES COVERED Final, 04/29/1999 – 09/29/2001		
4. TITLE AND SUBTITLE TIMEBENCH: A VISUAL ENVIRONMENT FOR THE DESIGN AND INTEGRATION OF OBJECT-ORIENTED REAL-TIME SYSTEMS			5. FUNDING NUMBERS C: F33615-99-C-1495 PE: 62173C PN: BMDI TA: SC WU: 02	
6. AUTHOR(S)				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) TIMESYS CORPORATION 4516 HENRY STREET PITTSBURGH, PA 15213			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) INFORMATION DIRECTORATE AIR FORCE RESEARCH LABORATORY AIR FORCE MATERIEL COMMAND WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334 POC: DR. STEVE HARY, AFRL/IFSC, (937) 255-4709 x4175			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-WP-TR-2001-1553	
11. SUPPLEMENTARY NOTES: THIS IS A SMALL BUSINESS INNOVATION RESEARCH (SBIR) PHASE II REPORT				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 Words) Phase II effort consisted of design, development, implementation, and testing of the software tools TimeStorm and TimeWiz for RoseRT. The TimeBench program funded, in its entirety, the TimeWiz for Rational RoseRT tool. As a result of this Phase II SBIR, the following capabilities have been developed: <ol style="list-style-type: none"> Working in conjunction with Information Directorate researchers under a SBIR Phase II contract and through other private funding, TimeSys Corporation has developed a commercial product called TimeWiz® for Rational RoseRT. This product is a customized version of the TimeSys TimeWiz product that offers significant analysis and synthesis capabilities to the users of the RoseRT modeling software. TimeWiz for RoseRT is described in more detail in Sections 4.1 and 5. TimeStorm is an integrated Development Environment (IDE) product that will be part of the TimeSys Linux operating system, making it simple to generate TimeSys Linux/RT applications for a full range of embedded platforms. The TimeStorm environment is designed to provide this full range of productivity, no matter what the resource level of the target. TimeStorm tools execute primarily on a host development PC, with shared access to a host-based dynamic linker and symbol table for a remote target system. TimeStorm is described in more detail in Sections 4.2 and 6. The Real-Time Foundation Class effort produced the Operating System Abstraction Layer (OSAL). The OSAL work is described in more detail in Section 2.3 and Appendix C. 				
14. SUBJECT TERMS Visual Design Tool, Schedulability Analysis, Component repository			15. NUMBER OF PAGES 68	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR	
NSN 7540-01-280-5500			Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102	

Table of Contents

1. INTRODUCTION	1
1.1 TIMEBENCH PROJECT SUMMARY	1
2. SUMMARY OF PHASE II WORK	3
2.1 DESIGN OF OBJECT-ORIENTED TIMING ATTRIBUTES AND RT-UML (TASK #1).....	3
2.1.1 <i>Description of Timing Model in TimeWiz for RoseRT</i>	4
2.2 CODE GENERATION AND REVERSE ENGINEERING (TASK #2).....	6
2.2.1 <i>Open C++</i>	6
2.2.2 <i>FLEX/Bison</i>	7
2.2.3 <i>ANTLR</i>	7
2.2.4 <i>SORCERER</i>	8
2.2.5 <i>MUSKOX v4.0</i>	8
2.2.6 <i>EDG C++ Front End</i>	8
2.2.7 <i>Compiler Resources</i>	9
2.2.8 <i>Comparison and Conclusion</i>	9
2.3 REAL-TIME FOUNDATION CLASSES (RTFC) OVERVIEW (TASK #3).....	10
2.4 DESIGN AND IMPLEMENTATION OF A COMPONENT AND ATTRIBUTE CATALOG AND API (TASK #4 AND TASK #6).....	11
2.4.1 <i>Synopsis</i>	11
2.4.2 <i>TimeWiz Property Extensions</i>	11
2.4.3 <i>TimeWiz API Extensions</i>	11
2.5 DESIGN AND INITIAL PROTOTYPING OF A FRIENDLY VISUALIZATION USER-INTERFACE	11
2.6 ARCHITECTURE OPENNESS (TASK #6)	12
3. SUMMARY OF PROGRAM ACCOMPLISHMENTS	13
4. COMMERCIALIZATION RESULTS	14
4.1 TIMEWIZ FOR RATIONAL ROSERT	14
4.2 TIMESTORM INTEGRATED DEVELOPMENT ENVIRONMENT (IDE).....	16
4.3 PRODUCT MARKETING PLAN	17
5. TIMEWIZ FOR RATIONAL ROSERT	19
5.1 OVERVIEW OF RATIONAL ROSERT	19
5.2 OVERVIEW OF TIMEWIZ FOR ROSERT	19
5.3 TIMEWIZ FOR RATIONAL ROSERT FEATURES.....	19
6. TIMESTORM	21
6.1 TIMESTORM OVERVIEW	21
6.2 TIMESTORM CAPABILITIES	21
6.3 CROSS-DEVELOPMENT WITH TIMESTORM	22
6.4 TIMESTORM FEATURES.....	22
APPENDIX A – TIMEBENCH STATEMENT OF WORK	24
APPENDIX B - RESULTS OF PHASE II WORK SUMMARIZED BY TASK	25
APPENDIX C – OSAL IMPLEMENTATION	27
C.1. TSAPERIODICTHREAD.....	27
C.2. TSCLOCK	28
C.3. TSCONDITION	29

C.4.	TSCONFIGDB	31
C.5.	TSDEBUG.....	33
C.6.	TSEVENT	34
C.7.	TSFILEDB.....	36
C.8.	TSGRABLOCK.....	37
C.9.	TSLOG	38
C.10.	TSMAPDWTODW	39
C.11.	TSMAPSTRINGTODW	42
C.12.	TSMEMLOG.....	43
C.13.	TSMUTEX.....	44
C.14.	TSNULLLOG	45
C.15.	TSPERIODICTHREAD	46
C.16.	TSRECURSIVESPINLOCK	47
C.17.	TSREF	48
C.18.	TSREFOBJECT	50
C.19.	TSSEMAPHORE	51
C.20.	TSSHARED BUDDY ALLOC.....	52
C.21.	TSSHARED MEMORY.....	54
C.22.	TSSPINLOCK	55
C.23.	TSSYSTEM.....	56
C.24.	TSTHREAD.....	57

FOREWORD

Version Notice

Document Release	Date	Revision Purpose
FR-Release-3.1	August 14, 2001	Final technical editing based on feedback from customer
FR-Release-3.0	August 14, 2001	Technical editing and review copy sent to customer
FR-Release-2.2	August 13, 2001	Added Appendix B and C
FR-Release-2.1	August 9, 2001	Addressed review comments from customer
FR-Release-2.0	May 23, 2001	Added Appendix A, Inserted new Section 3
FR-Release-1.3	May 7, 2001	Document baselined, and sent to customer.
FR-Release-1.2	May 4, 2001	Editing on section 2.
FR-Release-1.1	May 2, 2001	Wrote Sections 2, 4, 5.
FR-Release-1.0	April 30, 2001	Document created.

The current release of this document is considered a complete replacement of previous versions unless otherwise stated.

TimeSys Corporation. has made every effort to ensure that this document is accurate at the time of printing. Obtain additional copies of this document, as well as updated releases, from:

TIMEBENCH Project Manager
TimeSys Corporation
4516 Henry Street
Pittsburgh, PA 15213
(412) 681-6899

Final Report

TIMEBENCH

1. Introduction

This document represents the final technical report deliverable (CDRL A001) on the TimeBench contract F33615-99-C-1495.

1.1 TimeBench Project Summary

The TimeBench project started under contract F33615-98-C-1343 with the Air Force Research Laboratory (AFRL) as a result of a Phase I proposal TimeSys submitted against topic BMDO98-010. TimeSys, upon successful completion of the Phase I project was able to secure private sector investment funding, and thus submitted a Phase II proposal to BMDO and AFRL for consideration under the *FasTrack* program. In January 1999, TimeSys was notified that our Phase II proposal had been approved for negotiation; we started our Phase II contract (F33615-99-C-1495) in April 1999.

Table 1 summarizes our work on the entire (Phase I and II) TimeBench project. Note, throughout this final report we use the term “TimeBench” to refer to the SBIR project/contract and “TimeStorm and TimeWiz for RoseRT” as the software tools that have been developed under that project, which will be commercialized by TimeSys.

Project Phase	Period of Performance	Summary of Work
Phase I	Jun 98 – Dec 98 F33615-98-C-1343	We investigated, designed, and prototyped capabilities of a visual software workbench environment called TimeBench for designing, modeling, analyzing, reusing, and integrating object-oriented real-time systems.
Phase II	Apr 99 – May 01 F33615-99-C-1495	We developed and commercialized TimeStorm and TimeWiz for RoseRT, software workbench environments for designing, modeling, analyzing, reusing, and integrating object-oriented real-time systems. In addition, the Real-Time Foundation Classes (RTFC) was developed for the VxWorks, Win32 and RT-Mach operating systems; this product is currently not commercially available from TimeSys. We anticipate being able to bring this to market in the future. Within TimeStorm and TimeWiz for RoseRT, class hierarchies and timing information of real-time and embedded systems were represented visually. Object hierarchies and specified timing constraints were used to generate code for specific targets and programming languages. Subsystems can be coded incrementally while retaining the timing behavior of the final workload. Class

		<p>hierarchies were represented using real-time extensions to UML (Unified Modeling Language) resulting in RT-UML, which captured timing, scheduling and concurrency information in addition to relationships between subsystems, modules and classes. RT-UML representation was tightly integrated with the timing analysis capabilities of TimeWiz, a tool which applies rate-monotonic analysis techniques.</p> <p>TimeStorm and TimeWiz for RoseRT have (1) a highly visual and interactive interface, (2) a sophisticated diagramming utility to represent class hierarchies and timing information of real-time and embedded systems, (3) an auto-coding facility to generate code for specific targets and programming languages, (4) an incremental coding facility that allows subsystems to be coded incrementally while retaining the timing behavior of the final workload, (5) a reverse-engineering facility to re-generate visual and semantic information automatically from user code, and (6) a component repository that can store and retrieve reusable COTS and custom software components.</p>
--	--	--

Table 1 – Summary of TimeBench Project

2. Summary of Phase II Work

Phase II effort consisted of design, development, implementation, and testing of the software tools TimeStorm and TimeWiz for RoseRT. The TimeBench program funded, in its entirety, the TimeWiz for Rational RoseRT tool. As a result of this Phase II SBIR the following capabilities have been developed:

1. Working in conjunction with Information Directorate researchers under a SBIR Phase II contract and through other private funding, TimeSys Corporation has developed a commercial product called TimeWiz® for Rational RoseRT. This product is a customized version of the TimeSys TimeWiz product that offers significant analysis and synthesis capabilities to the users of the RoseRT modeling software. TimeWiz for RoseRT is described in more detail in Sections 4.1 and 5.
2. TimeStorm is an Integrated Development Environment (IDE) product that will be part of the TimeSys Linux Development Environment (LDE). TimeStorm has been developed especially to produce software for the TimeSys Linux/RT operating system, making it simple to generate TimeSys Linux/RT applications for a full range of embedded platforms. The TimeStorm environment is designed to provide this full range of productivity no matter what the resource level of the target. TimeStorm tools execute primarily on a host development PC, with shared access to a host-based dynamic linker and symbol table for a remote target system. TimeStorm is described in more detail in Sections 4.2 and 6.
3. The Real-Time Foundation Class effort produced the Operating System Abstraction Layer (OSAL). The OSAL work is described in more detail in Section 2.3 and Appendix C.

2.1 Design of Object-Oriented Timing Attributes and RT-UML (Task #1)

Within TimeWiz for RoseRT, the representation of the attributes along with associated objects is graphical, textual (as in a spread-sheet), and as modifiable component properties. Since we are actively participating in the Real-Time UML standards groups within OMG (Object Management Group) responsible for standardizing Real-Time specific extensions to UML, this effort will enable us to support the RT-UML standard product when the standard is released.

This task was a precursor to establish the scope of temporal model representation within object-oriented architectures. The entities, their relationships, and the relevant properties, which are integral part of the temporal model, need to be seamlessly represented within the object-oriented paradigm.

As part of this task we investigated and designed the timing attributes that will be associated with object-oriented system components in the object-oriented representation. Although the majority of the concepts already existed in TimeWiz, we made important additions to the modeling of event-driven and distributed systems. These involved the development of Trigger events that can be initiated from other actions (called Internal Events), and Tracer events which enable tracing pipelined architectures.

All the existing attributes were summarized and documented for future reference. This effort also partially contributed to the development of a supplemental user manual for TimeWiz. These attribute form stereotypes can be used within the UML. We have proposed this model with essential attributes for adoption as the Real-Time UML standard.

2.1.1 Description of Timing Model in TimeWiz for RoseRT

TimeWiz for RoseRT uses an object-oriented framework to capture and analyze the temporal model of a real-time system. A brief summary of this model is presented here, further details can be found in Chapter 3 of the TimeWiz for RoseRT user manual.

While there are many techniques for object modeling (e.g., Unified Modeling Language), temporal models have been built largely using the Software Engineering Institute (SEI) standard model described in [1]¹.

In the context of the TimeWiz for RoseRT tool, the following terms are used:

- *System architecture* describes the composition of the system in terms of system hardware architecture and system software architecture.
- *Hardware architecture* is described using hierarchy objects (which can contain other objects) and resources (which represent hardware elements capable of executing software ‘actions’).
- *Software architecture* is described using hierarchy objects (which can contain other objects) and events (which represent thread triggers, such as clock interrupts), and actions (which represent a sequence of executable code in a thread between scheduling points of interest).

2.1.1.1 Resources

Resources represent elementary (typically, hardware) objects on which actions execute. A resource can be a physical resource (a processor, network element, backplane, etc.), or a logical resource (a buffer, semaphore, or shared memory).

TimeWiz for RoseRT allows the resource to be any hardware component. The properties of the resource can be customized by a TimeWiz developer. The TimeWiz API (Application Programming Interface) can then be used to implement the analysis and simulation ‘plug-in’ specific for that resource.

Single node analysis for the “CPU” resource is supported in TimeWiz for RoseRT using the same API. The relevant resources in this context are the CPU and the Logical Resource.

2.1.1.2 CPU

The CPU resource represents a processor and possibly an operating system.

The CPU object may have zero, one, or more logical resources, or may execute zero, one, or more actions. Events may reference a CPU or Logical Resource.

The Logical Resource typically represents a shared resource, e.g., a buffer or semaphore. A logical resource is bound to a physical resource via the user-entered property “*Physical Resource*”. The data

¹ [1] A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems, from Software Engineering Institute and Carnegie-Mellon University by [Mark H. Klein](#), [Thomas Ralya](#), [Bill Pollak](#), [Ray Obenza](#), Kluwer Academic Publishers.

sharing policy for access to the logical resource is defined by the property *Data Sharing Policy* of the CPU.

2.1.1.3 Actions

Actions represent a single, sequential, executable segment of code that requires one physical resource and zero, one, or more logical resources. Also, Actions need to be “triggered” through an Event as part of the response to that Event. More than one event may trigger the same Action as part of its response.

An action that is not bound to a physical resource does not *execute* in the system. Also, an Action which is not in the response of any Event does not execute, even if it is bound to a Resource.

An action can only be bound to one physical resource. However, multiple logical resources can be used by an action. This restriction follows from the basic RMA model and the definitions of physical resources and actions and the ability to do schedulability analysis on physical resources.

An Action is bound to a resource via the property “*Executes On*” simply by selecting from the list of available CPUs, or by dragging and dropping the action on a CPU or by selecting automatic binding algorithms from the Analyze menu within TimeWiz.

Logical Resources may be specified for consumption by the Action via the property *Logical Resource List*. This property can be set to a string, a comma delimited list of names of the Logical Resources.

An action can be made part of a response to an event via the ‘*Response*’ property of the Event. This property is manipulated by connecting the event to the action in the software diagram view.

2.1.1.4 Events

Events in TimeWiz for RoseRT are Trigger Events.

Trigger Events represent a periodic (or aperiodic) scheduling of a sequence of actions on a single resource. On a single CPU resource, these represent the “arrival” or “ready state” of a thread or process; they may also represent the arrival of an interrupt, in response to which a sequence of actions is initiated.

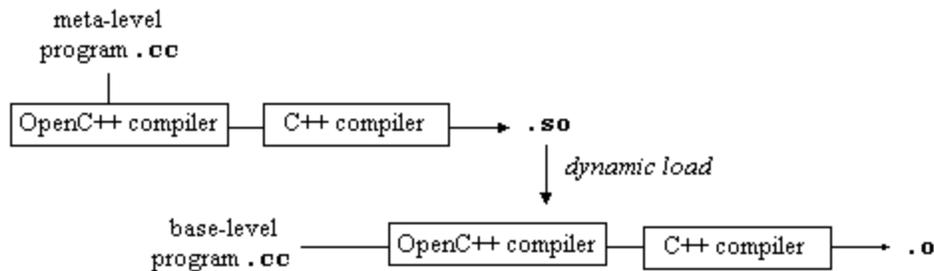
2.2 Code Generation and Reverse Engineering (Task #2)

Parsers and code generators to support this activity were investigated. The critical aspect of the parser/generator subsystem is a C++ parser that can parse C++ program into specified representations, extract specific tokens, constructs, and patterns. The following is a survey of available parsing options that were considered.

2.2.1 Open C++

OpenC++ is a toolkit for C++ translators and analyzers. It was designed to enable the users to develop those tools without concerning tedious parts of the development such as the parser and the type system. There are a number of tools that OpenC++ facilitates the development of. For example, the users can easily develop a C++ translator for implementing a language extension to C++ or for optimizing the compilation of their class libraries. Moreover, OpenC++ is useful to develop a source-code analyzer such as one for producing the class-inheritance graph of a C++ program.

The programmer who want to use OpenC++ writes a meta-level program, which specifies how to translate or analyze a C++ program. It is written in C++ and defines a small number of classes. Then the meta-level program is compiled by the OpenC++ compiler and (dynamically or statically) linked to the compiler itself as a compiler plug-in. The resulting compiler translates or analyzes a source program (it is called a base-level program for distinction) as the meta-level program specifies.



The meta-level program is written according to the programming interface called the OpenC++ MOP (Metaobject Protocol.) Through this interface, the internal structure of the compiler is exposed to the programmers with object-oriented abstraction.

The base-level program is first preprocessed by the C++ preprocessor, and then divided into small pieces of code. These pieces of code are translated by class metaobjects and assembled again into a complete C++ program. In the OpenC++ MOP, the pieces of code is represented by Ptree metaobjects in the form of parse tree (that is, linked list). Although the metaobjects are identical to regular C++ objects, they exist in the compiler and represent a meta aspect of the base-level program. This is why they are not simply called objects but metaobjects.

The class metaobject is selected according to the static type of the translated piece of code. For example, if the piece of code is a member call on a Point object:

```
p0->move(3, 4)
```

Then it is translated by the class metaobject for Point (the type of p.) It is given to the class metaobject in the form of parse tree and translated, for example, into this:

```
(++counter, p0->move(3, 4))
```

This translation is similar to the one by Lisp macros, but it is type-oriented. The translation by the metaobjects is applied not only a member call but also other kinds of code involved with the C++ class system, such as data member access and class declaration.

The programmer who wants to customize the source-to-source translation writes a meta-level program to define a new class metaobject. This class metaobject is associated with a particular class in the base-level program and controls the translation of the code involved with the class. Thus, the translation is applied only to the particular class and the rest of the code involved with the other classes remains as is.

The class metaobject can use other aspects of the base-level program during the source-code translation. In addition to the parse tree, it can access the semantic information such as static types and class definitions. These various aspects of the program facilitates the implementation of complex source-code translation and analysis. Furthermore, the OpenC++ MOP enables syntax extensions so that the base-level programmers can write annotations to help the translation or the analysis.

The meta architecture of OpenC++ might look very different from the architecture of other reflective languages. However, note that the class metaobject still controls the behavior of the base-level objects, which are instances of the class. The uniqueness of the OpenC++ MOP is only that the class metaobject does not interpret the base-level program in the customized way, but rather translates that program at compile time so that the customized behavior is implemented. The readers will find that, as in other reflective languages, the class metaobject has a member function for every basic action of the object, such as member calls, data reading/writing, object creation, and so forth, for customizing the object behavior.

2.2.2 FLEX/Bison

<http://www.monmouth.com/~wstrett/lex-yacc/lex-yacc.html>

flex is a tool for generating scanners: programs which recognized lexical patterns in text. flex reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. flex generates as output a C source file, lex.yy.c, which defines a routine yylex(). This file is compiled and linked with the -lfl library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

Bison is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar. Once you are proficient with Bison, you may use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.

Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change. Anyone familiar with Yacc should be able to use Bison with little trouble. You need to be fluent in C programming in order to use Bison or to understand this manual.

2.2.3 ANTLR

Features: ANTLR constructs human-readable recursive-descent parsers in C or C++ from predicated-LL($k > 1$) grammars. Many context-sensitive languages and languages requiring infinite look-ahead are recognizable with ANTLR parsers. Can automatically build AST's. Has new and powerful error recovery mechanism.

Distribution: Complete C source code, totally public domain. Free at site <ftp://ftp.parr-research.com/pub/pccts/>.

Platform: Any platform that compiles C or C++.

2.2.4 SORCERER

AST transformer / walker (source-to-source translation)

Features: A SORCERER grammar describes AST content and structure. You may annotate the grammar with actions to effect a translation or manipulate the tree itself. Generates recursive-descent tree walkers in C or C++ (soon Java). Same flavor/syntax as ANTLR. Not tied to a parser generator or any other tool.

Distribution: Complete C source code, totally public domain. Free at site <ftp://ftp.parresearch.com/pub/pccts/sorcerer/>.

Platform: Any platform that compiles C or C++.

2.2.5 MUSKOX v4.0

C++ and Java Parser Generator

WEB: <http://www.mastersys.com>

Features: Annotates C++ & Java classes and interfaces with EBNF LR(K) grammars.
Grammar inheritance and redefinition of rules.
Multiple parsers, recording/replay of trace logs, HTML pretty-printing.
Default and user-defined lexer and error processing.
Syntactic look-aheads, semantic predicates, syntax trees.

Distribution: Parser Generator executable.
Runtime Framework C++ & Java source.
Downloadable from the web site.
Documentation in Postscript and Adobe PDF formats.

Platform: PC Windows 95 & NT.

2.2.6 EDG C++ Front End

Compiler Front End

WEB: <http://www.edg.com>

Features: Does full syntax and semantic analysis on C++ source code, producing an AST-like internal representation. Accepts most of the modern features of the language, e.g., templates, exceptions, RTTI, new-style casts, array new/delete, namespaces, member templates. Also

accepts ANSI/ISO C, several older dialects of both C and C++, and Microsoft C and C++ extensions.

Distribution: Source code and internal documentation

Platform: Portable; has been used on all major Unix platforms, plus Windows NT/95.

2.2.7 Compiler Resources

OO Lexer and Parser Generator for C++ and Class Library

Yacc++ and the Language Objects Library

Phone 1 (508) 435-5016 Fax 1 (508) 435-4847

Features: Outputs C++ classes of lexers and parsers and optionally for tokens, non-terminals, and rules as specified in the grammar. Regular expressions integrated with BNF, LALR & LR, grammar inheritance. Library support for various AST, input, error, and symbol table classes.

Distribution: The Yacc++ generators are shipped as executables (sources available) and the Language Objects Library is shipped as C++ source code and pre-compiled for supported targets. Numerous examples and makefiles are included. Printed manuals include Installation, Tutorial, and Reference Guides.

Platform: Windows NT/95 Microsoft Visual C++

2.2.8 Comparison and Conclusion

	OpenC++	ANTLR	Flex/Biso	Sourcerer	Musko	EDG	Compiler Resources
Platform	Win	Win	Win	Win	Win	Win	Win
Source code available	Yes	Yes	Yes	Yes	Yes	Yes	No
Cost	Free	Free	Free	Free	Free	Yes	Yes
Royalties	None	No	No	No	No	Yes	Yes
Support for other languages	Yes	No	Yes	No	No	Yes	No
Ease of use	Great	Fair	Fair	Fair	Fair	Good	Good

Open C++ was chosen due to its ease of use, support for multiple languages, and cost and source code availability.

We did not have to make any modifications to core Open C++ parser, although we needed to implement its interface with the rest of the TimeStorm program.

2.3 Real-Time Foundation Classes (RTFC) Overview (Task #3)

The Real-Time Foundation Classes serve as an uniform interface for the code generators for different platforms. They present an abstraction above the operating system layer that applications can invoke. Since the TimeBench project was specifically intended to be platform neutral, this abstraction is an essential component of the finished effort.

We implemented the System Services component of the Real-Time Foundation Classes as the Operating System Abstraction Layer (OSAL). The OSAL consists of an uniform API and a set of platform specific Adaptors. The API provides System Services and Debug Services. Platform specific Adaptors hide the operating system details for the specific operating systems they are targeted to.

Broad categories of OSAL services are as follows:

- Threads
- Events
- Clocks
- Timers
- Mutex
- Semaphores
- Shared Memory

OSAL Adaptors were implemented for VxWorks, Windows, and RT-Mach.

When the strategic decision was made to utilize the Rational RoseRT as the modeling base for the timing analysis, the original RTFC implementation was rendered unusable, since the API layer and modeling layers needed extensive redesign, which was commercially impossible.

In addition, Rational RoseRT includes its own set of libraries which are platform neutral. Therefore, we decided to utilize the Rational RoseRT primitives to achieve the intended purpose of supporting multiple platform with an uniform modeling interface and being able to perform a timing analysis.

Appendix C provides an index of the original TimeSys OSAL implementation.

2.4 Design and Implementation of a Component and Attribute Catalog and API (Task #4 and Task #6)

2.4.1 Synopsis

The majority of the attribute-value model was already developed earlier as part of TimeWiz effort. As part of the TimeBench project, we developed extensions to our existing attribute-value model that can express relationship among objects in addition to their properties. We also developed extensions to the API that allow parsing of the object relationships.

2.4.2 TimeWiz Property Extensions

In the basic TimeWiz property-value model, objects have properties. These properties can be custom defined using attributes that can be specified for each property. Each property has a type, which can be numeric or string.

The new property types of CTWZObject and CTWZObjList were added to our existing model. The former is used to contain a reference to another object and the latter is used to contain references to an array of objects.

An example of CTWZObject's use would be an object referencing another object. For example, if an Action references a physical resource, the CTWZObject property of the Action is set to the Physical Resource it references.

An example of the CTWZObjList's use would be an object referencing another set of objects. For example, when a physical resource references a set of actions, the CTWZObjList property of that physical resource is set to the array containing the actions.

2.4.3 TimeWiz API Extensions

API extensions were created to allow enhanced hierarchical navigation. The following specific functions were created as part of the TimeBench project:

1. GetParent: Gets the parent object of an object
2. GetFirstChild: Gets the first child of the parent object
3. GetNextChild: Gets the next child object of the parent object
4. GetFirstConnection: Gets the first connection of an object
5. GetNextConnection: Gets the next connection of an object

2.5 Design and Initial Prototyping of a friendly Visualization User-Interface

This task completed the initial prototype of the highly visual, friendly and hierarchical interface within TimeStorm and TimeWiz for RoseRT. An interactive demonstration was available and was presented at the Phase I status meeting.

Further details of the user interface, menus, dialogs, and the interaction are documented in the TimeStorm and TimeWiz for RoseRT user manuals.

As part of this TimeBench project, we designed and developed the entire user interface in TimeStorm, including its advanced features of Class Diagrams and Activity Diagrams, which are not yet available in the retail version.

None of the TimeWiz for RoseRT user interface was developed as part of this program since that portion was derived from TimeWiz, developed previously.

2.6 Architecture Openness (Task #6)

In support of Task #6, the plug-in architecture was designed whereby external modules could operate as plug-in units. An open API extension was designed and documented as part of Task #3. Part of the implementation was completed in Phase I, with the remaining portion completed in Phase II.

3. Summary of Program Accomplishments

- Commercialization and full product release of the TimeStorm (May 2001) Integrated Development Environment (IDE) tool. The TimeStorm tool is currently bundled as part of the TimeSys Linux Development Environment (LDE).
- Commercialization and full product release (November 2000) of the TimeWiz for RoseRT. See www.timesys.com/news/pr111000.html.
- Worked with Program Manager and Lockheed Martin to develop a SBIR success story for the TimeBench project.
- Created and delivered product demonstrations for both TimeStorm and TimeWiz for RoseRT to a variety of commercial and defense contractors.
- Delivered technical presentations regarding our TimeWiz for RoseRT at conference such as: Embedded Systems Conference and LinuxWorld.
- Based on our tool development activities, participated and contributed to the Real-Time UML standards groups within OMG responsible for standardizing Real-Time specific extensions to UML.
- Designed and implemented the System Services component of the Real-Time Foundation Classes as the Operating System Abstraction Layer (OSAL). Created runtime bindings to VxWorks, WinNT, WinCE, and Real-time Mach.

4. Commercialization Results

The TimeBench program funded in its entirety, the development and commercialization of the TimeWiz for Rational RoseRT tool.

4.1 TimeWiz for Rational RoseRT

Working in conjunction with Information Directorate researchers under a SBIR Phase II contract and through other private funding, TimeSys Corporation has developed a commercial product called TimeWiz® for Rational RoseRT (see Figure 1). This product is a customized version of the TimeSys TimeWiz product that offers significant analysis and synthesis capabilities to the users of the RoseRT modeling software.

A TimeWiz design/analysis model consists of models of resource architecture, software architecture, and a mapping of software architecture elements to the resources. TimeWiz can be used in the early stages of architecture analysis to model the real-time software architecture of a system, by focusing only on aspects relevant for performance modeling. At later stages, when a model of software architecture has been built in Rational RoseRT, it can be automatically imported into TimeWiz. The software architecture modeling also allows users to specify the timing requirements and assumptions for analysis purposes.

The analysis and synthesis engine gives users the ability to assess whether a TimeWiz design model can meet the response-time requirements specified. The analysis engine also gives users worst-case response times (under the given workload assumptions), and provides feedback when response time requirements cannot be met. By changing various properties, users can also perform “what-if” analysis. The synthesis engine of the tool computes scheduling attributes (priorities) for elements in the design model in order to meet the response-time requirements (whenever possible). The synthesis engine can also be used to optimize the mapping from logical threads to physical threads.

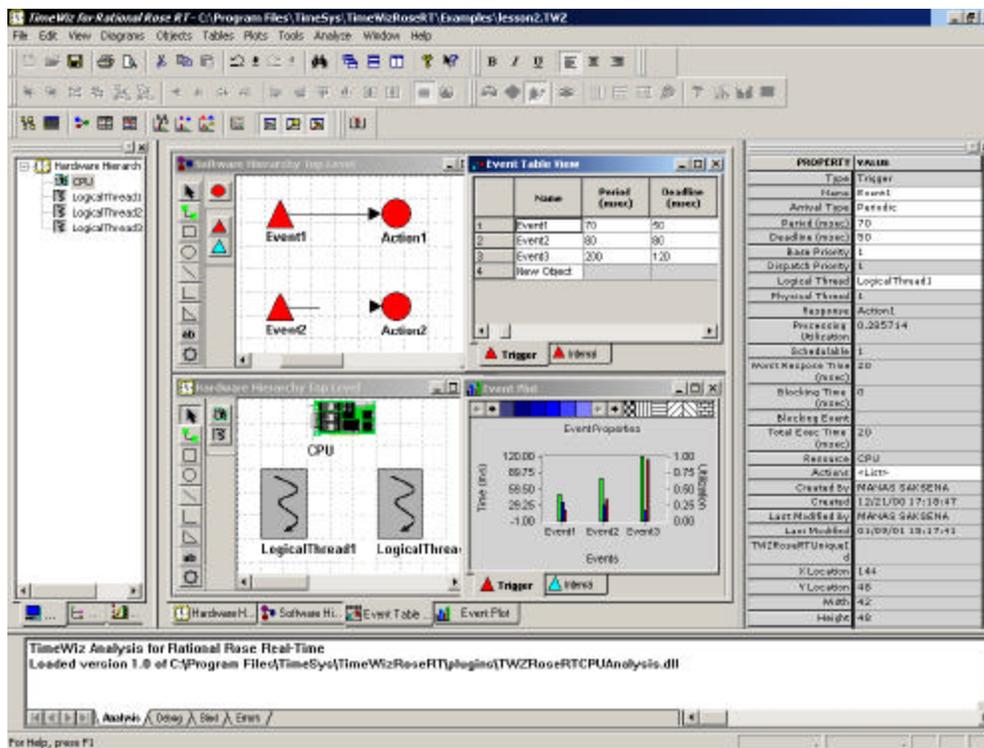


Figure 1 – Screen shot of TimeWiz® for Rational RoseRT tool

TimeWiz® for Rational Rose RealTime is designed to work in conjunction with RoseRT during the entire development cycle, thus providing a complete solution for UML based modeling, design, automatic code generation, and timing analysis for developing timing sensitive embedded systems.

The analysis and synthesis engine in TimeWiz for Rational RoseRT is based on proven rate-monotonic analysis (RMA) techniques that have been specifically extended for addressing response-time requirements of time-critical scenarios in a Rational RoseRT design model. This technology development work is based on several years of research done by TimeWiz researchers aimed at extending rate-monotonic techniques to take advantage of Rational RoseRT's superior capabilities for modeling, design, and automated implementation of complex event-driven software.

TimeWiz for RoseRT provides a unique combination of UML-based design and temporal analysis capabilities such as:

- Support for RMA analysis for UML models with an underlying event-driven execution paradigm.
- Support for synthesis and optimization of implementation attributes/parameters based on RMA design. The synthesized implementation attributes feed into code-generation engine.
- An approach for interfacing between a UML design tool, and an RMA analysis/synthesis tool.

4.2 TimeStorm Integrated Development Environment (IDE)

Working in conjunction with Information Directorate researchers under a SBIR Phase II contract and through other private funding, TimeSys Corporation has also developed a commercial product called TimeStorm (see Figure 2) based on the innovative development of the prototype software in Phase II on the TimeBench contract F33615-99-C-1495.

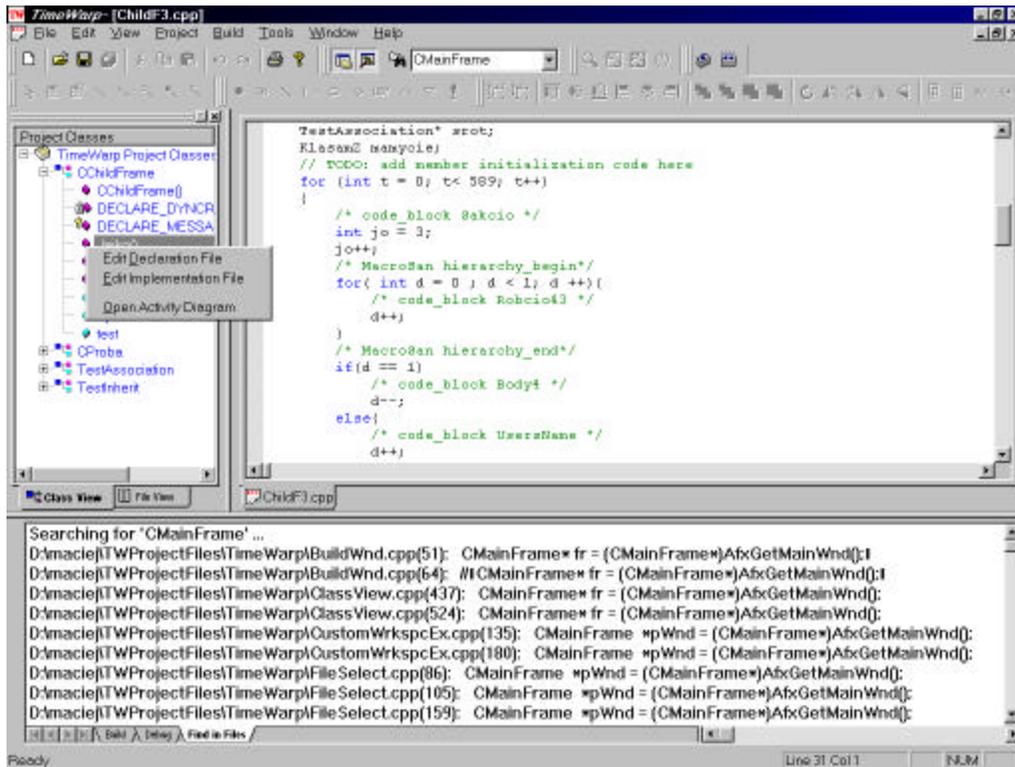


Figure 2– Screen shot of TimeStorm Tool

TimeStorm is an Integrated Development Environment (IDE) product that will be part of the TimeSys Linux Development Environment (LDE). TimeStorm has been developed especially to produce software for the TimeSys Linux/RT operating system, making it simple to generate TimeSys Linux/RT applications for a full range of embedded platforms. The TimeStorm environment is designed to provide this full range of productivity no matter what the resource level of the target. TimeStorm tools execute primarily on a host development PC, with shared access to a host-based dynamic linker and symbol table for a remote target system.

4.3 Product Marketing Plan

TimeSys is a leading vendor of a predictable, real-time, open source based software development and runtime platform that is robust enough to be used in used in complex, real-time, multi-function, convergence devices and systems. Our broad range of solutions incorporate:

- an open, run-time platform (based on Linux) with guaranteed real-time performance
- a standard middleware platform based on Java that provides predictable real-time performance
- a strong set of interoperable tools addressing real-time software development and software architectural design and analysis
- expert training, mentoring and consulting services

We sell and market our products/services into the following markets:

- **Telecommunication/Networking** – telecommunications and data infrastructure equipment such as switches and routers for service providers, enterprise network equipment, remote access concentrators, and mobile phones/pagers, etc.

Telecommunication/networking is the largest segment (30% share) of the current embedded systems market, and is still growing at over 20% compounded annually. The major telecommunication/networking equipment manufacturers (e.g., Cisco, Lucent, Alcatel, Nortel, 3Com) have been competing fiercely, trying to differentiate themselves by delivering more robust functionality to the Service Providers (e.g., AT&T, Sprint, Verizon, WorldCom) as well as reducing their prices and delivery times. This had precipitated a move to open source software, and also the development of more complex, real-time, multi-function, convergence devices and systems. All of these trends represent an excellent opportunity to sell TimeSys products/services because of our unique value-add provided by our predictable, real-time, open source based software development and runtime platform, and by our Partitioned Virtual Machine (PVM) solution. This segment is also attractive to us because the “product to market” success rate here (>90%) is considerable higher when compare with the Consumer Electronics segment (75%) for example.

- **Defense/Aerospace** – commercial and military aircraft, satellite systems, radar/sonar systems, C3, missile guidance, avionics, simulators, etc.

Defense/Aerospace continues to be a large segment (2nd largest at 16% share) of the current embedded systems market. This segment has strict requirements for predictable, real-time systems, and thus, represents an excellent fit for TimeSys products/services, especially as major weapon system programs migrate to use of open source and COTS products.

- **Industrial Automation** – manufacturing and process control systems, motion controllers, operator interfaces, robotics, etc.

Industrial Automation is also a large segment (3rd largest at 10% share) of the current embedded systems market. Process control systems have requirements for predictable, real-time behavior in the context of controlling enterprises such as power, manufacturing, and chemical plants. This segment is starting to move away from expensive, proprietary software systems, and thus, represents an excellent opportunity for selling TimeSys products/services.

- **Consumer Electronics** - set-top boxes, multimedia (Internet) appliances, home networking gateways, gaming systems, automotive entertainment systems, etc.

Consumer Electronics is a rapidly growing segment of the current embedded systems market (over 50% CAGR), which builds small memory footprint, multi-function, convergence devices. The requirement for open source solutions is very strong in this segment because of “time to market” and “cost of development” business drivers. This segment will provide good business opportunities for TimeSys because of our unique value-add provided by our predictable, real-time, open source platform and our Partitioned Virtual Machine (PVM) solution. However, these opportunities have to be well qualified for us because the project cancellation rate is considerably high (over 25%) in this market segment.

- **Automotive** – electronic control units in chassis systems, powertrain electronics, body electronics/security systems, in-vehicle information/computing systems, etc.

Automotive is also a rapidly growing segment of the current embedded systems market (over 30% CAGR). The in-vehicle information/computing systems (i.e., a multi-function, convergence device) piece of this market is growing at over 75% CAGR. As with the Industrial Automation segment, the first-tier automotive subsystem suppliers are starting to move towards COTS solutions and away from expensive, proprietary products. This segment is also a strong believer in the use of software modeling tools such as our TimeWiz tool. All of these trends represent an excellent opportunity to sell TimeSys products/services because of our unique value-add provided by our predictable, real-time, open source platform, our Partitioned Virtual Machine (PVM) solution, and our real-time software design, analysis, and modeling tools.

We estimate that the addressable market size for TimeSys solutions is \$2.3B in 2003, while the estimated size in 1999 was \$1.1B, representing a 20% annual growth in size.

- Market size for Real-Time Operating Systems products is expected to be \$1.2B in 2003, with TimeSys addressable market size of \$360M, representing the projected share for the Linux based RTOS's. The product addressing this market is TimeSys Linux.
- Market size of large real-time software solutions (including telecommunications infrastructure, automotive and defense areas) is \$10B in 2003, with TimeSys addressable market size of \$1.5B, representing outsourced software solutions provided by TimeSys. The products addressing this market are our SuiteTime family of tools, packaged solutions and turnkey systems.
- Market size for the new embedded devices is expected to be \$3.8B, with TimeSys addressable market size of \$500M, representing real-time and QoS aspects of these devices. The products addressing this market are JTime and TimeSys Linux.

TimeSys Corporation's product strategy is to provide the best integrated bundle of tools for real-time system design, analysis, development, and deployment. We want to become the de facto real-time Linux vendor of choice in our target markets.

In the Defense/Aerospace market, our senior staff members have an excellent and long-standing reputation for expertise in real-time systems. To date, TimeSys products such as TimeBench and TimeWiz for RoseRT have been used by prime contractors for many Navy and Air Force programs including the DD-21, AWACS, B1-B, BSY submarine program, the F-16 and F-22 programs. Our customer base in this market includes leaders such: Lockheed Martin, Raytheon, Boeing, and BAE Systems.

5. TimeWiz for Rational RoseRT

TimeWiz™ for Rational RoseRT is a tool for the design and analysis of real-time systems. It is based on extended rate-monotonic analysis for addressing response-time requirements of time-critical scenarios in a Rational RoseRT design model. The analysis and synthesis engine in TimeWiz for Rose RT is based on several years of research done by TimeWiz technology experts aimed at extending rate-monotonic techniques to take advantage of Rational RoseRT's superior capabilities for modeling, design, and automated implementation of complex event-driven software.

5.1 Overview of Rational RoseRT

Rational RoseRT™ from Rational Software, is a UML-based design and development environment for developing embedded real-time software. Developers can use RoseRT to design their applications using a mix of UML models and C or C++ code and then generate executable code using automatic code generators and in-built support for language compilers and target environments. It allows a flexible many-to-one mapping from active objects (capsules) to threads - allowing the designer to trade off implementation overheads with the desired level of preemptibility needed to meet response time requirements. In addition, it allows events to be prioritized to meet response time requirements.

5.2 Overview of TimeWiz for RoseRT

TimeWiz for Rational RoseRT is designed to be used in conjunction with Rational Rose Real-Time throughout the design and development lifecycle. In the early stages of architectural analysis, TimeWiz for Rational RoseRT can be used to get early feedback on the feasibility of meeting the system timing requirements. It can also be used to quickly analyze the impact of different architectures on the ability of the system to meet its timing requirements, thus facilitating the selection of an appropriate architecture. All this can be done before any detailed behavioral modeling is undertaken.

During later stages of development, as more details are added to the system structure and behavior, TimeWiz for Rational RoseRT can be used to continually assess the impact of design choices on the ability of the system to meet the timing requirements, and giving valuable feedback to the designers when performance bottlenecks are identified.

TimeWiz for Rational RoseRT is also designed to extend the automatic code generation capabilities of RoseRT by automatically synthesizing design attributes such as event and thread priorities, as well as mapping the logical design model to threads. TimeWiz for RoseRT eliminates the guesswork that must be used by designers in assigning values to these attributes - it not only analyzes the impact of these attributes on response times, but also synthesizes values for these attributes to meet specified response time requirements.

5.3 TimeWiz for Rational RoseRT Features

TimeWiz for Rational RoseRT features include:

- **Modeling** - a TimeWiz design/analysis model consists of models of hardware or resource architecture, software architecture, and a mapping of software architecture elements to the resources. TimeWiz can be used in the early stages of architecture analysis to model the real-time software architecture of your system, by focusing only on aspects relevant for performance modeling. At later stages, when a model of software architecture has been built in Rational

RoseRT it can be automatically imported into TimeWiz. The software architecture modeling also allows you to specify the timing requirements and assumptions for analysis purposes.

- **Analysis and Synthesis** - the analysis and synthesis engine gives you the ability to assess whether a TimeWiz design model can meet the response-time requirements you specify. The analysis engine gives you worst-case response times (under the given workload assumptions). It also gives you feedback when response time requirements cannot be met. By changing various properties, you can also perform "what-if" analysis. The synthesis engine of the tool computes scheduling attributes (priorities) for elements in the design model in order to meet the response-time requirements (whenever possible). The synthesis engine is particularly useful to automatically generate an optimal mapping from the design's logical concurrency architecture to a physical concurrency architecture in the implementation.
- **Visualization of Results** - the results of TimeWiz analysis and synthesis can be visualized using action, event, and resource plots and tables, which come in a wide variety of styles and can be customized as needed.
- **Documentation** - the software and hardware architecture diagrams, as well as the analysis and synthesis results, can be documented in reports. These reports include software and hardware diagrams, tables, and plots, and can be viewed onscreen or printed.
- **Integration with Rational RoseRT** - TimeWiz for RoseRT is integrated with Rational Rose Real-Time to allow you to develop your model in Rational RoseRT, then export the model to TimeWiz, and re-import the modified properties back into Rational RoseRT

6. TimeStorm

6.1 TimeStorm Overview

TimeStorm is an integrated development environment (IDE) from TimeSys Corporation. TimeStorm has been developed primarily to produce software for the TimeSys Linux operating system, making it simple to generate TimeSys Linux applications for a full range of embedded platforms. TimeStorm interactive development tools include:

- An integrated source-code editor
- A customizable project management facility
- Integrated C and C++ compilers and make
- A source-level debugger
- Support for downloading and executing applications on a remote target

The TimeStorm IDE is designed to run on a Windows host environment, thus enabling the development of applications for Linux targets with modest resources that is typical in an embedded system. It supports a seamless interaction with the application target. With TimeStorm, concepts can pass from idea to implementation very quickly. Fast incremental downloads of application code are linked dynamically with the TimeSys Linux/RT operating system and are thus available for symbolic interaction with minimal delay. The TimeStorm development environment offers a unified perspective that integrates design, development, and analysis.

6.2 TimeStorm Capabilities

TimeStorm runs on a host machine running Windows NT, using the gnu compiler tools as a backend for compiling source files into target executables. It then automatically exports these applications to custom embedded targets running TimeSys Linux and helps ensure that they execute. A project settings wizard enables you to specify target platform compilation, as well as linking and exporting properties, for each individual project. TimeStorm's support of projects means that an entire group of files can be compiled into a single executable. Additionally, TimeStorm offers support for makefiles, allowing you to specify the interrelationships between the files in a project.

TimeStorm allows you to write and edit source code for TimeSys Linux programs in C and C++. It offers complete editing features, including code-specific highlighting and search-and-replace capabilities. Two control trees make manipulating code easier and more efficient. One of these trees indexes all the classes, methods, and variables used in a project; clicking on a list item takes you to where that item is declared or implemented and adding an item to the list will cause its source code to be automatically generated. The other tree lists every file in the project, allowing you to quickly locate and select files to open.

Additionally, TimeStorm makes use of the Unified Modeling Language (UML) to aid in visualizing systems. TimeStorm's UML-based features center on two different UML diagrams, the class diagram and the flow-chart activity diagram. A class diagram describes the characteristics of and relationships between the classes used in a program, while a flow-chart activity diagram models every step between the initiation and the completion of a task. With TimeStorm, you can draw class and activity diagrams from scratch, generate them by reverse-engineering C++ code, and use them to generate code. TimeStorm also keeps the diagrams and the code in sync so that, if you edit one, the other will change accordingly.

6.3 Cross-Development with TimeStorm

The philosophy behind TimeStorm is that different operating systems excel at different things, and programs should be designed so that each operating system does what it is best at. TimeSys Linux is designed especially for running embedded systems with time constraints, while Windows NT is one of the most prevalent operating systems for software development. Therefore, we have created a development environment for TimeSys Linux in which Windows NT handles the editing and compilation tasks, exporting the completed executable to TimeSys Linux so that all it needs to worry about is the actual running of the program.

TimeStorm lets you edit, compile, link, and store application code using only the Windows NT host. Even once the code is exported to the target, the host machine still controls its running and debugging. To understand the TimeStorm environment more clearly, it is helpful to describe the typical development process.

The hardware in a typical development environment includes one or more networked development host systems and one or more embedded target systems. A number of means exist for connecting the target and host systems, but the most popular options are Ethernet or serial links.

A typical development system is allotted large amounts of RAM and disk space, backup media, printers, and other peripherals. In contrast, a typical target system has only the resources required by the real-time application, and perhaps a small amount of additional resources for testing and debugging. The target may comprise no more than a CPU with on-chip RAM and a serial I/O channel.

Application modules in C or C++ are compiled with the cross-compiler provided as part of TimeStorm. These application modules can draw on the TimeStorm run-time libraries to accelerate application development. A fundamental advantage of the TimeStorm environment is that the application modules do not need to be linked with the run-time system libraries or even with each other. Instead, TimeStorm can load the re-locatable object modules directly, using the symbol tables in each object module to resolve external symbol references dynamically. In TimeStorm, this symbol table resolution is done by the target server (which executes on the host).

Because the application does not need to be linked fully, object modules during development are considerably smaller in TimeStorm than they would be in other environments. This is a major advantage in a cross-development cycle – the smaller the downloads, the shorter the development cycle. Dynamic linking means that it makes sense for even partially completed modules to be downloaded for incremental testing and debugging. The host-resident TimeStorm debugger can then be used interactively to invoke and test either individual application routines or complete tasks.

TimeStorm maintains a complete host-resident symbol table for the target. This table is incremental: as it downloads each individual object module, it incorporates the symbols from that module. Thus, you can use the original symbolic names to perform a host of activities, like examining variables, calling subroutines, spawning tasks, disassembling code in memory, setting breakpoints, or tracing subroutine calls.

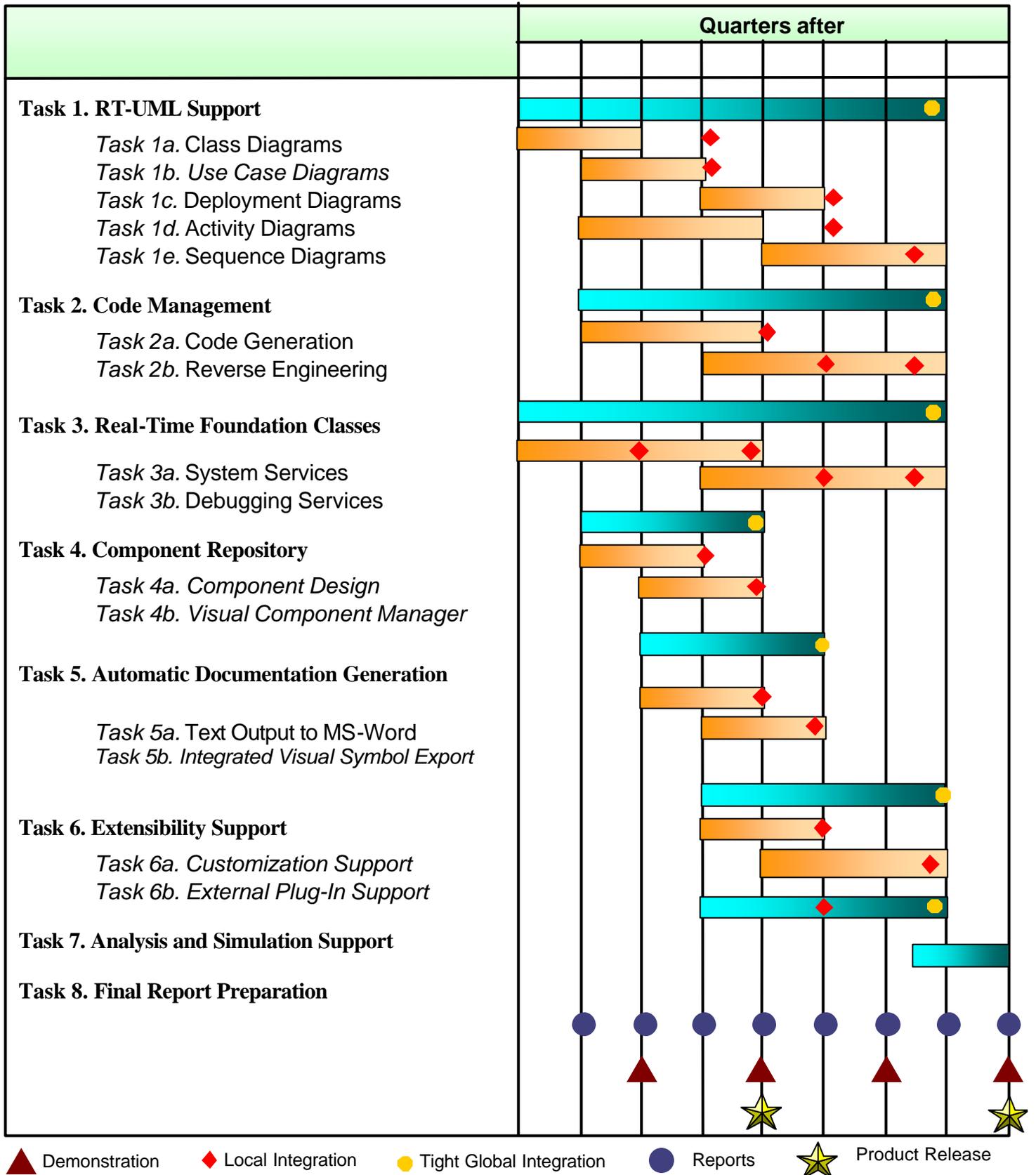
The TimeStorm IDE also includes a debugger, which allows developers to view and debug an application's original source code. Setting breakpoints, single-stepping, examining structures, and other similar tasks are all done at the source level, using a convenient graphical interface.

6.4 TimeStorm Features

TimeStorm features include:

- Full ability to export programs to a variety of embedded systems running TimeSys Linux/RT
- Project Settings Wizard to specify target platform compilation and set linking and exporting properties
- Control trees for ease of navigation — view lists of every class, method, and variable used in a project, or of every file currently open. Create classes and methods from within the tree.
- Unified Modeling Language (UML)-based visualization features
- Powerful editor with search-and-replace and code-specific highlighting features
- Multiple targets per project — use the same project to build different versions of your code, such as debug and release builds, or projects aimed at different processors
- Threaded execution — continue working with the editor while a program compiles in the background
- The GNU Toolkit, a high-performance 32-bit cross-development compiler for C and C++. make and other supporting programs are also included.

Appendix A – TimeBench Statement of Work



Appendix B - Results of Phase II Work Summarized by Task

1. Task 1. RT-UML Support
 - TimeStorm (advanced version) supports class diagrams and activity diagrams.
 - Since the TimeWiz analysis modules were integrated with Rational RoseRT, Use Case, Deployment, and Sequence diagrams are available in Rational Rose and therefore supported in the final integrated TimeWiz for RoseRT product.

2. Task 2. Code Management
 - Code Generation and Reverse Engineering capabilities were prototyped in the advanced version of TimeStorm.

3. Task 3. Real-Time Foundation Classes
 - *Task 3a. System Services:*
 - We implemented the System Services and Debugging Services library modules as documented in Section 0. However, since the analysis modules were integrated with Rational RoseRT, these services ,which are also available in Rational RoseRT, were used in the final TimeWiz for RoseRT product.

4. Task 4. Component Repository
 - TimeWiz for Rational RoseRT contains customized palettes with module objects required for timing analysis. These had to utilize the TimeWiz Catalog Designer, which serves as a form of component design tool.
 - The originally envisioned Visual Component Manager was not completed.

5. Task 5. Automatic Documentation Generation
 - Since the TimeWiz analysis modules were integrated with Rational RoseRT, the Text Output to MS-Word and Integrated Visual Symbol Export features are available in Rational Rose and therefore are supported in the final TimeWiz for RoseRT product.

6. Task 6. Extensibility Support

- Since the TimeWiz analysis modules were integrated with Rational RoseRT, the Customization Support and External Plug-In Support features are available in Rational Rose and therefore are supported in the final TimeWiz for RoseRT product.

7. Task 7. Analysis and Simulation Support

- The TimeWiz analysis modules were integrated with Rational RoseRT. The Simulation support modules were not integrated with Rational RoseRT.

Appendix C – OSAL Implementation

C.1. tsAperiodicThread

Header

tsThread.h

Parent

tsThread

Children

None

Methods

```
static tsAperiodicThread create(tsThreadFunc pFunc, void *pArg);
```

Description

A class wrapper for aperiodic threads.

Notes

The function passed in to create must have this signature:

```
uint32 (*userfunc)(void *);
```

Methods

```
static tsAperiodicThread create(tsThreadFunc pFunc, void *pArg);
```

Creates an aperiodic thread. The thread runs immediately. The user passes in a function of type `tsThreadFunc` and an argument it wishes to pass to the function. The argument may be `NULL`. The thread is assigned the default system priority - the application may change the priority with the [priority](#) call.

C.2. tsClock

Header

[tsClock.h](#)

Parent

none on Win32 and WinCE, tsRefObject on RT-Mach

Children

None

Methods

```
static tsClock create();  
tsTimeSpec currentTime();  
uint64 frequency() const;
```

Description

tsClock is a timer relative to the boot of the machine. It is a high-frequency timer.

Notes

Methods

```
static tsClock create();
```

Creates a clock.

```
tsTimeSpec currentTime();
```

Returns the current system time. tsTimeSpec is a platform-specific way of returning relative times. Note that the time returned by currentTime is relative to the boot of the system and should not be interpreted as an absolute time or date.

```
uint64 frequency() const;
```

Returns the frequency of the clock in ticks per second.

C.3. tsCondition

Header

[tsCondition.h](#)

Parent

tsRef

Children

None

Methods

```
static tsCondition create(const char *pscName=NULL);  
bool wait(tsMutex);  
bool signal();
```

Description

An implementation of pthread-style condition variables. Condition variables require a mutex and an external condition to check. The mutex must be locked during calls to wait and signal. Condition variables can be process-local or process-shared.

Notes

A typical use of condition variables (error checking code removed for brevity:

```
#include "tsMutex.h"  
#include "tsCondition.h"  
tsMutex g_mutex; // assume these are initialized somewhere else  
tsCondition g_cond;  
bool g_fSomeCondition;  
void waiter()  
{  
    g_mutex.lock();  
    while(!g_fSomeCondition)  
    {  
        // the while loop is necessary to handle spurious wakeups  
        g_cond.wait(g_mutex);  
    }  
    g_mutex.unlock();  
}  
void waitee()  
{  
    // do some stuff  
    ...  
    // now we are ready  
    g_mutex.lock();  
        g_cond.signal();  
    g_mutex.unlock();  
}
```

Methods

```
static tsCondition create(const char *pscName=NULL);
```

Creates a condition object. If pscName is NULL, then a process local object is created. If pscName is not NULL, then the system creates a process-shared object with the name pscName. If create fails, then the returned tsRef will be NULL. The last error can be retrieved with tsLastError.

```
bool wait(tsMutex);
```

Waits for the condition variable to be signaled. The passed in mutex must be locked before calling this method. On some systems, waits can be woken up spuriously, even if the associated condition has not been met. Therefore it is important to enclose calls to wait in a while loop that checks the value of the associated condition (see above). A return value of false indicates an error, use tsLastError to check the error value.

```
bool signal();
```

Signals the condition variable that the associated condition is now true. The associated mutex with the condition variable must be locked, and must be the same mutex that is being waited on (see above). A return value of false indicates an error, use tsLastError to check the error value.

C.4. tsConfigDB

Header

[tsConfigDB.h](#)

Parent

None

Children

tsFileDB

Methods

```
virtual void clear()=0;
virtual void read()=0;
virtual void write()=0;
virtual void category(const char *pscCategory)=0;
virtual const char *category() const=0;
virtual bool query(const char *pscKey, const char *&value)=0;
virtual bool set(const char *pscKey, const char *pscValue)=0;
virtual bool erase(const char *pscKey)=0;
virtual bool erase()=0;
```

Description

An abstract interface for a configuration database made of categories and key/value pairs. The category is a namespace for various string keys and their string values. This configuration database is used internally for per-file debug logging.

Notes

None

Methods

```
virtual void clear()=0;
```

Clear the entire configuration database.

```
virtual void read()=0;
```

Read the configuration database from the data source.

```
virtual void write()=0;
```

Write the configuration database to the data source.

```
virtual void category(const char *pscCategory)=0;
```

Set the current category. If the category does not exist, the database will create it.

```
virtual const char *category() const=0;
```

Query the current category.

```
virtual bool query(const char *pscKey, const char *&value)=0;
```

Query the value of a key in the current category. If the key does not exist, false will be returned.

```
virtual bool set(const char *pscKey, const char *pscValue)=0;
```

Set the value of a key in the current category. If the key does not exist, the key will be created. If an error occurs, false will be returned.

```
virtual bool erase(const char *pscKey)=0;
```

Erase a key in the current category. If the key does not exist, false will be returned.

```
virtual bool erase()=0;
```

Erase the entire current category.

C.5. **tsDebug**

Header

[tsDebug.h](#)

Parent

None

Children

None

Methods

no public methods

Description

tsDebug wraps the debugging interface. Currently all public exported functionality is provided through the macros `TS_DEBUG_LOG` and `TSTRACE`.

Notes

None

Methods

C.6. tsEvent

Header

[tsEvent.h](#)

Parent

tsRef

Children

None

Methods

```
static tsEvent create(bool fSet, bool fAutoReset=true, const char
*pscName=NULL);
bool created() const;
bool set();
bool reset();
bool wait();
```

Description

Creates a Win32-style event object which can be used for synchronization. Unlike condition variables, events require no external mutex. Events come in two types - manual and auto. Manual events require reset to be called manually for the event to go from the set to the reset state. Auto events automatically go into reset state as soon as any waiting threads have been resumed.

Notes

None

Methods

```
static tsEvent create(bool fSet, bool fAutoReset=true, const char
*pscName=NULL);
```

Create an event object. Creation options include creating the object set (fSet=true), or creating an auto event (fAutoReset=true). Events can be process-local (pscName=NULL) or process-shared (pscName=some name). If creation fails, a NULL tsRef will be returned and the last error status can be queried with tsLastError.

```
bool created() const;
```

Returns true if this particular object created the internal referenced event, or false if it didn't and someone else had created it first. Note this is only relevant for process-shared events. Process-local events always return true.

```
bool set();
```

Set the event. Any waiting threads will be released. If the event is an auto-reset event, after waiting threads have been released the event will be reset. Returns false in case of an error, and last error status can be queried with `tsLastError`.

```
bool reset();
```

Resets the event. Returns false in case of an error, and last error status can be queried with `tsLastError`.

```
bool wait();
```

Causes the current thread to wait for the event to be set. If the event is in the set state already, the calling code will return immediately. If the event is in the reset state, the calling thread will be suspended until the event is set. Returns false in case of an error, and last error status can be queried with `tsLastError`.

C.7. tsFileDB

Header

[tsConfigDB.h](#)

Parent

tsConfigDB

Children

None

Methods

```
tsFileDB(const char *pscFile);
```

Description

A concrete implementation of the tsConfigDB interface. It reads and writes configuration information from a file.

Notes

The configuration file format has two header lines and then lines following with [categories] in brackets and key=value pairs.

```
TimeSys Config Database
Version=1.0
[Sections]
threadtest.cpp=false
[Debug]
Debug=true
```

Methods

```
tsFileDB(const char *pscFile);
```

Creates a file database object that reads and writes from the passed file.

C.8. tsGrabLock

Header

[tsSpinLock.h](#)

Parent

None

Children

None

Methods

```
tsGrabLock(tsSpinLock &spl);  
~tsGrabLock();
```

Description

A convenience class that grabs a spinlock's lock in the constructor and releases it in the destructor.

Notes

Sample use:

```
tsSpinLock g_someLock;  
{  
    tsGrabLock lock(g_someLock); // lock grabbed  
    // do stuff  
    ...  
} // destructor called, lock released
```

Methods

```
tsGrabLock(tsSpinLock &spl);
```

Grabs the spinlock.

```
~tsGrabLock();
```

Releases the spinlock.

C.9. tsLog

Header

[tsLog.h](#)

Parent

None

Children

tsMemLog

tsNullLog

Methods

```
virtual void log(const char *pscFormat, ...)=0;
```

Description

Abstract interface for log output.

Notes

None

Methods

```
virtual void log(const char *pscFormat, ...)=0;
```

Supports printf-style arguments. Sends formatted output to a logical logging device.

C.10. tsMapDwToDw

Header

[tsCol.h](#)

Parent

None

Children

None

Methods

```
tsMapDwToDw(uint32 mapsize = DEF_MAP_SIZE);
~tsMapDwToDw();
bool setAt(uint32 key, uint32 value);
bool lookup(uint32 key, uint32 &value);
uint32 remove();
bool removeAt(uint32 key);
void removeAll();
position firstAssoc() const;
position lastAssoc() const;
uint32 nextAssoc(position &pos, uint32 &key) const;
uint32 prevAssoc(position &pos, uint32 &key) const;
bool isEmpty() const;
uint32 numItems() const;
bool isValidPos(position pos) const;
```

Description

An generic hash table class. It is recommend you use the MAKE_DWMAP and MAKE_SIZED_DWMAP macros, instead of using this class directly. The macros create "poor man's template" versions of the maps that cast from specified types automatically.

Notes

Iteration over the map example:

```
tsMapDwToDw map;
position pos = map.firstAssoc();
while(pos != NULL)
{
    uint32 value, key;
    value = nextAssoc(pos, key);
    // do something with key/value
}
```

Methods

```
tsMapDwToDw(uint32 mapsize = DEF_MAP_SIZE);
```

Create a hash table with DEF_MAP_SIZE entries. DEF_MAP_SIZE should be a prime number. This class does not grow the hash table, so performance will degrade when there are more than mapsize elements stored in the table.

```
~tsMapDwToDw();
```

Destroy the hashtable. Note that if the values are pointers to structures, the destructor does not free up their memory.

```
bool setAt(uint32 key, uint32 value);
```

Set the value at a key. If there already exists a value at that key, setAt will return false. To set the value of a key with an existing value, use removeAt first, then setAt.

```
bool lookup(uint32 key, uint32 &value);
```

Lookup the value at a key. If the key is not in the map, it will return false, and the contents of value is undefined. If the key is in the map, value will contain the value.

```
uint32 remove();
```

Remove an arbitrary value in the map and return the value. The hash table is unordered so there is no way of knowing which key/value pair will be removed.

```
bool removeAt(uint32 key);
```

Remove the key/value pair from the map. If the key does not exist, false will be returned.

```
void removeAll();
```

Remove all key/value pairs from the map.

```
position firstAssoc() const;
```

Return a generic position value for the first association in the map, for iteration over the map. Iterations are not guaranteed to be in any specific order. If the map is empty, NULL will be returned.

```
position lastAssoc() const;
```

Return a generic position value for the last association in the map, for reverse iteration over the map. Iterations are not guaranteed to be in any specific order. If the map is empty, NULL will be returned.

```
uint32 nextAssoc(position &pos, uint32 &key) const;
```

Iterate the next key/value pair given a position. Position will be updated for the next element, key will contain the current key, and nextAssoc will return the value at that key. The position may become NULL after this call, this indicates the current key/value pair is the last pair in the map.

```
uint32 prevAssoc(position &pos, uint32 &key) const;
```

Reverse iterate the next key/value pair given a position. Position will be updated for the previous element, key will contain the current key, and prevAssoc will return the value at that key. The position may become NULL after this call, this indicates the current key/value pair is the last pair in the map.

```
bool isEmpty() const;
```

Returns true if the map is empty, false if it is not.

```
uint32 numItems() const;
```

Returns the number of key/value pairs in the map.

```
bool isValidPos(position pos) const;
```

Returns true if the passed position is valid, false if not.

C.11. tsMapStringToDw

Header

tsCol.h

Parent

None

Children

None

Methods

```
tsMapStringToDw(uint32 mapsize = DEF_MAP_SIZE);
~tsMapStringToDw();
bool setAt(const char *key, uint32 value);
bool lookup(const char *key, uint32 &value);
uint32 remove();
bool removeAt(const char *key);
void removeAll();
position firstAssoc() const;
position lastAssoc() const;
const char *keyAt(position pos) const;
uint32 nextAssoc(position &pos) const;
uint32 prevAssoc(position &pos) const;
bool isEmpty() const;
uint32 numItems() const;
bool isValidPos(position pos) const;
```

Description

An generic hash table class for mapping strings to dwords (32 bit values). Strings are duplicated with strdup for storage in the class. It is recommend you use the MAKE_STRMAP and MAKE_SIZED_STRMAP macros, instead of using this class directly. The macros create "poor man's template" versions of the maps that cast from specified types automatically.

Notes

See tsMapDwToDw for a description of class methods.

tsMapStringToDw iteration is slightly different. An example:

```
tsMapStringToDw map;
position pos = map.firstAssoc();
while(pos != NULL)
{
    const char *pscKey = map.keyAt(pos);
    uint32 value = map.nextAssoc(pos);
    // do something with key/value
}
```

Methods

```
const char *keyAt(position pos) const;
```

Returns the key at the current position.

C.12. tsMemLog

Header

tsLog.h

Parent

tsLog

Children

None

Methods

```
tsMemLog(const char *pscLogName, uint32 logSize);  
void wait();  
void readLock(const char *&p, uint32 &nbytes);  
void readUnlock();
```

Description

A concrete implementation of the tsLog interface. Log output is written to a shared-memory circular buffer. If the buffer is full, old data is overwritten. Methods are provided for reading from the shared buffer.

Notes

The logdebug util app will read from the default shared memory circular set up by OSAL and print it to either stdout or to a file.

Methods

```
tsMemLog(const char *pscLogName, uint32 logSize);
```

Initialize the memory log. Open the shared memory region denoted by pscLogName. The shared memory region will be of size logSize. Note that all clients of the same shared memory region must agree on logSize!

```
void wait();
```

Wait for data to be available for reading.

```
void readLock(const char *&p, uint32 &nbytes);
```

Read available data. When this function returns, the circular buffer will be locked and a pointer to the data to be read will be in p, and the number of bytes to read will be in nbytes. Make your processing in between calls to readLock and readUnlock fast, as writers are prevented from writing to the shared memory region while it is locked. For example, instead of saving to a file from these buffers directly, copy the buffers into temporary storage first and then unlock the buffer. Then write from your temporary buffer to a file.

```
void readUnlock();
```

Unlock the read buffer. Once you have unlocked data, the span of data returned by readLock will be lost forever!

C.13. tsMutex

Header

tsMutex.h

Parent

tsRef

Children

None

Methods

```
static tsMutex create(const char *pscName=NULL);  
bool lock();  
bool unlock();  
bool created() const;
```

Description

Creates a mutual exclusion synchronization objects. Mutexes in OSAL implement the priority inheritance protocol regardless of platform. Mutexes can be either process-local or process-shared.

Notes

Mutex locks are freed if the process crashes or terminates unexpectedly.

Methods

```
static tsMutex create(const char *pscName=NULL);
```

Create a mutex object. Mutexes can be either process-local (`pscName=NULL`) or process-shared (`pscName=some name`). If creation fails, the returned `tsRef` will be `NULL`. Call `tsLastError` to retrieve the error value.

```
bool lock();
```

Lock the mutex. If the lock fails, `false` will be returned and error information can be queried with `tsLastError`.

```
bool unlock();
```

Unlock the mutex. If the mutex can not be unlocked, `false` will be returned and error information can be queried with `tsLastError`.

```
bool created() const;
```

Returns `true` if this particular object created the internal referenced mutex, or `false` if it didn't and someone else had created it first. Note this is only relevant for process-shared mutexes. Process-local mutexes always return `true`.

C.14. tsNullLog

Header

tsLog.h

Parent

tsLog

Children

None

Methods

None

Description

A concrete implementation of the [tsLog](#) interface. Log output goes nowhere and results in a no-op.

Notes

None

Methods

C.15. tsPeriodicThread

Header

tsThread.h

Parent

tsThread

Children

None

Methods

```
static tsPeriodicThread create(tsThreadFunc pFunc, void *pArg,  
tsTimeSpec period);  
bool period(tsTimeSpec ts);  
tsTimeSpec period() const;
```

Description

Wrappers for periodic threads.

Notes

Methods

```
static tsPeriodicThread create(tsThreadFunc pFunc, void *pArg,  
tsTimeSpec period);
```

Creates a periodic thread with period specified by the third argument. If creation fails, the returned tsRef will be NULL, and error status can be queried with tsLastError.

```
bool period(tsTimeSpec ts);
```

Set the thread's period on the fly. Returns false on error, error status can be retrieved with tsLastError.

```
tsTimeSpec period() const;
```

Query and return the thread's period. If an error occurs a tsTimeSpec struct with -1 seconds and -1 nanoseconds is returned and error status can be retrieved with tsLastError.

C.16. tsRecursiveSpinLock

Header

tsSpinLock.h

Parent

None

Children

None

Methods

```
void grab();  
void release();
```

Description

A spin lock that can be locked recursively. tsRecursiveSpinLock maintains a recursion count, so that repeated calls to lock from the same thread do not deadlock. Note that repeated lock calls must be balanced by a same number of unlock calls.

Notes

tsRecursiveSpinLock has a higher overhead than tsSpinLock, so use it only in cases that are absolutely necessary.

Methods

See the definitions of tsSpinLock's methods. The interface is the same, the only difference is recursion will not cause deadlock.

C.17. tsRef

Header

tsRef.h

Parent

None

Children

tsClock (RT-Mach only), tsCondition, tsEvent, tsMutex, tsSemaphore,
tsSharedBuddyAlloc, tsSharedMemory, tsThread

Methods

```
tsRef();  
tsRef(tsRefObject *pRef);  
tsRef(const tsRef & ref);  
~tsRef();  
tsRef &operator=(const tsRef &ref);  
bool operator==(const tsRef &ref) const;  
bool operator<(const tsRef &ref) const;  
bool isNull() const;
```

Description

A envelope object. tsRef wraps the nastiness of reference counting internal implementation objects, freeing the user from that burden. It also insulates the implementation of various OSAL library classes from their interface. tsRef objects should always be passed by value or reference, but tsRef *'s should never be passed.

Notes

None

Methods

```
tsRef();
```

Constructs a NULL tsRef object.

```
tsRef(tsRefObject *pRef);
```

Constructs a tsRef object given a tsRefObject to point to. Adds a reference to pRef.

```
tsRef(const tsRef & ref);
```

Copy constructor. Adds a reference.

```
~tsRef();
```

When a tsRef object is destroyed, it removes the reference it had.

```
tsRef &operator=(const tsRef &ref);
```

Assigns a tsRef object to the current object. Keeps track of references.

```
bool operator==(const tsRef &ref) const;
```

Compares two tsRefs. If they point to the same object, then returns true.

```
bool operator<(const tsRef &ref) const;
```

Less than comparison on two tsRefs. Uses a pointer less than comparison.

```
bool isNull() const;
```

Returns true if this tsRef is NULL. Used mainly to check for error conditions during object creation.

C.18. tsRefObject

Header

tsRef.h

Parent

None

Children

internal classes

Methods

```
void addRef();  
void release();
```

Description

Base class for our letter objects. OSAL makes use of the envelope-letter pattern, where there are many smaller envelope objects that wrap around fewer large letter objects. This hides the often platform-specific details of the internal implementations from the clients using the envelope (tsRef-derived classes). It also automatically handles reference counting (a nice side benefit).

Notes

addRef and release are thread-safe.

Methods

```
void addRef();
```

Add a reference to the current object. Increments the internal reference count.

```
void release();
```

Release a reference to the current object. If the reference count becomes zero, delete the object and associated resources.

C.19. tsSemaphore

Header

tsSemaphore.h

Parent

tsRef

Children

None

Methods

```
static tsSemaphore create(int initVal, const char *pscName=NULL);
bool wait();
bool post();
```

Description

Implement a semaphore. Semaphores can be process-local or process-shared.

Notes

Currently OSAL's semaphores do not handle priority inversion on any platform (yet).

Methods

```
static tsSemaphore create(int initVal, const char *pscName=NULL);
```

Create a semaphore object. The object is initialized to a count of `initVal`, and is process-local (`pscName=NULL`) or process-shared (`pscName = some name`). If creation fails, the returned `tsRef` will be `NULL`. Error information can be queried with `tsLastError`.

```
bool wait();
```

Try to decrement the semaphore value. If the value is zero, block until value is increased. This is commonly known as the P() operation. Returns false if an error arises, use the `tsLastError` function to query the error information.

```
bool post();
```

Increment the semaphore's value. If any threads are blocking on the semaphore, wake one up. This is commonly known as the V() operation. Returns false if an error arises, use the `tsLastError` function to query the error information.

C.20. tsSharedBuddyAlloc

Header

tsMemory.h

Parent

tsRef

Children

None

Methods

```
static tsSharedBuddyAlloc create(void *pBase, uint32 offset, uint32
size, uint32 minAllocSize, bool fCreate);
void *base() const;
void *alloc(size_t size);
bool free(void *p);
bool fromPool(void *p) const;
bool empty() const;
bool lock(void *p);
bool unlock(void *p);
```

Description

A memory allocator for shared memory regions. It uses the buddy system to efficiently allocate memory chunks of different sizes.

Notes

None

Methods

```
static tsSharedBuddyAlloc create(void *pBase, uint32 offset, uint32
size, uint32 minAllocSize, bool fCreate);
```

Create a shared memory allocator. pBase describes the base address of the shared memory region. Offset is the offset into the shared memory region the pool should start at. size is the size of the buddy pool. minAlloc is the minimum allocation size to be supported - it should be at least 8. fCreate indicates whether the buddy pool should be initialized or just connected to. Returns a NULL tsRef if creation fails, error information can be retrieved with tsLastError.

```
void *base() const;
```

Returns the base address of the shared memory region (note: not of the buddy pool!).

```
void *alloc(size_t size);
```

Allocates a block big enough to hold size bytes. If the allocation fails it will return NULL.

```
bool free(void *p);
```

Frees up the block associated with p. P should be a pointer returned from alloc. If p is not from this pool, false will be returned.

```
bool fromPool(void *p) const;
```

Returns true if the passed point is inside the buddy pool.

```
bool empty() const;
```

Returns true if there are no allocations in the buddy pool.

```
bool lock(void *p);
```

Unimplemented - wire down the page that allocation is in.

```
bool unlock(void *p);
```

Unimplemented - unwire the page that allocation is in.

C.21. tsSharedMemory

Header

```
tsMemory.h
```

Parent

```
tsRef
```

Children

```
None
```

Methods

```
typedef enum
{
    ReadWrite,
    ReadOnly
} Flags;
static tsSharedMemory create(uint32 initSize, uint32 maxSize, Flags
flags, const char *pscName);
void *base() const;
uint32 maxSize() const;
```

Description

A named shared memory region. Shared memory regions, by definition, are process-shared. Memory may not be mapped at the same address in all processes, so data stored in shared memory should use no absolute references. OSAL provides a shared memory allocator, `tsSharedBuddyAlloc`, for easy management of shared memory. Combined with the macros `tsPointerToOffset` and `tsOffsetToPointer`, shared memory data structures are manageable.

Notes

Memory inside the shared memory region is must be committed/decommitted with the `tsVAlloc` and `tsVFree` calls. Address space should not be allocated with those calls, though! All clients of the same shared memory region must agree on the `maxSize` parameters!

Methods

```
static tsSharedMemory create(uint32 initSize, uint32 maxSize, Flags
flags, const char *pscName);
```

Creates a shared memory object. The object's total address space used is `maxSize` bytes, whereas only the first `initSize` bytes are actually committed to physical memory. The remainder can be managed with `tsVAlloc` or `tsVFree` - or maintained by `tsSharedBuddyAlloc` (the preferred method). The flags can be used to protect the memory as read/write or read-only. The name is required, it identifies the shared memory region. If creation fails, `create` will return a NULL `tsRef`. Error information can be queried with `tsLastError`.

```
void *base() const;
```

Return the base address of the shared memory region.

```
uint32 maxSize() const;
```

Return the maximum size of the shared memory region.

C.22. tsSpinLock

Header

tsSpinLock.h

Parent

None

Children

None

Methods

```
void grab();  
void release();
```

Description

A non-blocking synchronization object. tsSpinLock uses atomic operations to try and grab a mutual exclusion variable. If the variable is already taken, tsSpinLock will spin. Spinning consists of repeatedly polling the exclusion variable whilst sleeping for an exponentially growing amount of time. This is similar to the back-off algorithm of Ethernet. While it is not theoretically guaranteed to eliminate starvation, in practice it works very well.

Notes

tsSpinLock can not handle recursion. The same thread grabbing a spinlock twice will cause deadlock. Use tsRecursiveSpinLock to handle these situations.

Methods

```
void grab();
```

Atomically grab the spinlock. Spins until lock has been grabbed.

```
void release();
```

Release the spinlock. Note: does not check the make sure current thread owned the lock in the first place.

C.23. tsSystem

Header

tsSystem.h

Parent

None

Children

None

Methods

None

Description

Static methods for initializing/exiting OSAL, and some internally used utility routines. Nothing public exported as of yet.

Notes

None

Methods

None

C.24. tsThread

Header

tsThread.h

Parent

tsRef

Children

tsAperiodicThread, tsPeriodicThread

Methods

```
bool isPeriodic();
void sleep(uint32 ms);
void priority(uint16 pri);
uint16 priority() const;
static tsThread current();
```

Description

Base thread wrapper class, provides functionality common to periodic and aperiodic threads.

Notes

Max priority = 255, min priority = 0. Higher priorities are favored as far as scheduling goes.

Methods

```
bool isPeriodic();
```

Returns true if this thread is periodic and can be safely cast into a tsPeriodicThread object. If false, it can be safely cast into a tsAperiodicThread object.

```
void sleep(uint32 ms);
```

Sleep for ms milliseconds.

```
void priority(uint16 pri);
uint16 priority() const;
```

Change/retrieve the thread's base priority.

```
static tsThread current();
```

Retrieve the currently running thread.