TECHNISCHE UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK

The Design and Implementation of TOPSYS

Version 1.0

Thomas Bemmerl, Arndt Bode,

Peter Braun, Olav Hansen,

Thomas Treml, Roland Wismüller

July 1991

Contents

Preface

Arndt Bode

The development of an integrated, hierarchical tool environment and a parallel programming library for parallel systems has been started 1987 by the parallel processing group within the Institut für Informatik of Technische Universität München. A first version of TOPSYS (TOols for Parallel SYStems) and MMK (Multiprocessor Multitasking Kernel) have been implemented in 1988 on an iPSC Sugarcube (node processors: i286/287). Especially under the contract of Sonderforschungsbereich 0342, TOPSYS and MMK were ported to an iPSC/2 (node processors: i386/387) and later to an iPSC/860 (node processor: i860). Parts of the system have also been transferred meanwhile to multiprocessor systems based on T800 transputers and MC68030 microprocessors.

Since TOPSYS and MMK are not only being used by the researchers of SFB 0342 and a number of installations have been made all over the world, it was necessary to distribute written material on how to use and modify the system. User's guide and user's reference manual as well as a series of introductory articles on the system have been distributed as reports number 342/9/90A, 342/25/90A and 342/26/90A. The following document has been compiled to describe the details on the design and implementation of TOPSYS.

As TOPSYS and MMK do not aim at being commercial systems but rather a set of tools open to the sciencific community, the parallel processing group would appreciate to get feedback of any potential user of the tools in the sense of hints for bugs, additional features, extensions etc. Any kind of comment is particulary welcome!

Munich, June 1991

Arndt Bode

The Design and Implementation of TOPSYS

Thomas Bemmerl, Peter Braun, Olav Hansen, Thomas Treml, Roland Wismüller

This document is a description of the TOPSYS tools environment. Scope of the description is to give a detailed insight into the structure of these software packages. Using this document, a reader should be able to understand and extend the available system. This means, he can understand and modify the source code of TOPSYS by using the design documents and comments within the source code. The design documents are structured according to the hierarchical layered tool model of the TOPSYS system. In part 1 an introductory description explains the basic concepts and design decisions. The programming library MMK is presented as the basic parallel programming model. After this overview the layers of TOPSYS are explained from top to bottom according to the hierachical tool model. Part 2 gives details on the common graphical user interface used by all the tools of the TOPSYS environment. In part 3 of the document, the three interactive tools DETOP, PATOP and VISTOP developed so far are explained. Part 4 finally describes the structure of the last layer implemented on the host workstation, the Universal Tool Layer UTOL. This subsystem is common to all tools and monitors and implements the transformations of the source level to the machine level and vice versa. Part 5 of the document concentrates on the layers implemented in each processing node of the multiprocessor. The distributed software monitoring system is presented within this section. Part 6 briefly describes the source file structure and the commands needed to compile TOPSYS. Finally the appendices give exact descriptions of all the interfaces defined within the TOPSYS hierarchical tool model.

Part 1:

Introduction and Basic Concepts

Thomas Bemmerl

This chapter is aimed towards a description of the framework of the TOPSYS project as well as an introduction of the basic concepts and mechanisms. It is subdivided into three subchapters. The first one gives a motivation for the project and a general overview. In subchapter 2 and 3 we describe the MMK and TOPSYS from a conceptual point of view. The MMK section is a preconditioning for the understanding of the rest of the document. This subchapter explains the principles of the parallel programming model all applications developed with TOPSYS are based on. For a more detailed description of the MMK functionality, refer to the MMK User's Manual. The TOPSYS section of part 1 focuses on relations between the several subsystems and layers. In particular the interactions between the layers of the hierarchical tool model are explained by describing the execution of representative functions.

1 Motivation and General Overview

1.1 Motivation

Massively parallel systems promise to be the only possibility to increase computing performance in the future by several orders of magnitude. The recognition of this fact has lead in recent years to intensified research in the field of parallel and distributed systems [Jes89], [Mae86], [Gil88], [Bod91a]. Most of the research projects were concentrated on hardware aspects. The results of these projects have shown that parallel machines are adequate for increasing computing performance. An industrial outcome of this research work are massively parallel computer systems based on various architectures. At the moment, most available parallel computers are used for specific types of applications such as numerics or artificial intelligence. A reason for this limited use is the difficult useability and programmability of parallel systems. Most of the parallelization of applications on these machines has to be done by hand from computer science specialists. G. Papadopoulos from MIT comments this fact with the sentence: “It appears to be easier to build parallel machines than to use them” [Pap87]. To open a wider market for parallel computers, it is necessary to simplify their useability and programmability.

Therefore it is time to increase not only the performance of parallel systems but also to do research on programmability and useability of these types of computers. In the long run, parallel systems will be successful if they are as easy to use and to program as sequential computers. A first step toward this direction is to increase programmer productivity with adequate tools for configuration, specification, programming, validation and analysis of parallel applications. Adequate tools for all phases of the “software life cycle” for parallel systems have to be taken into account. From our point of view, tools for the first phases of the design cycle will not be very different from tools for sequential systems. The differences will increase more and more, with the development of tools for the late phases of the design cycle like validation and analysis. Available parallel machines especially lack in tools for observing the dynamic behaviour of the parallel execution run. Parallel debuggers, test systems, performance analyzers [Bur88], [Hof87] and graphical animation tools belong to this type of tools.

Apart form these monitoring and analysis tools, the programmer of a parallel computer needs in particular support for the mapping of the logical program structure onto the physical parallel architecture. This class of tools is represented by static or interactive specification, configuration and mapping tools [Gua89]. For the long term future a remapping and dynamic loadbalancing component has to be integrated into distributed operating systems to balance the load of the parallel machine during execution [Li89], [Art89], [Bod91b]. In order to simplify program creation, the process model offered by the operating system must be easy to use by the programmer. This requires a flexible and dynamic creation of processes as well as transparent and global operations on all processes, regardless on which processor node they are located.

Tools and system software mentioned in the last paragraphs are only partly available on existing parallel systems. In particular an integrated methodology for using the different tools during parallelization of applications is still missing. This fact gave us the major motivation to develope and implement an integrated tool environment for using and programming parallel computers within the TOPSYS project [Bem88]. The actual implementation of the TOPSYS environment was done on an iPSC/2 Hypercube with 32 processor nodes. Currently a port to iPSC/860 is under work.

1.2 Major Goals of the TOPSYS Project

In general, the major intention of the TOPSYS project is to simplify the usage and the programming of parallel systems for widening the range of applications for these types of machines. Massively parallel computers should become as easy to use as today's conventional (sequential) workstations. Therefore the first goal of TOPSYS is to work out a general methodology for using and programming parallel architectures and applications. Within the project we try to support the application of this methodology by the implementation of an integrated tool environment for all the phases of the "parallel software developement cycle".

In particular, we develope all the tools mentioned in the previous chapter as there are the distributed operating system kernel MMK [BeL90] with integrated dynamic load balancing, the

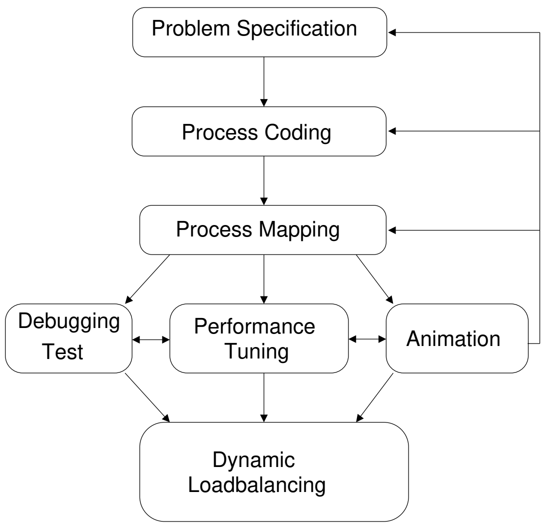


Fig. 1.1: Interactive TOPSYS methodology

specification and mapping tool SAMTOP, the parallel debugger DETOP, the performance analyzer PATOP [BHL90] and the graphical animation tool VISTOP. Figure 1.1 explains the interactive usage of the TOPSYS tools for programming parallel systems.

The second goal of the TOPSYS project is motivated by the large number of different available parallel architectures. The range of parallel architectures varies from tightly coupled systems with shared memory and fine grain parallelism to loosely coupled distributed memory machines with coarse grain parallelism based on message passing. None of these proposed architectures became the "industry standard" during the previous decade. In contrary a lot of different architectures are in use and they differ with respect to the node processor, the node architecture (board) and the communication network. This fact gives the motivation to design the TOPSYS environment very portable, flexible and adaptable with respect to:

the processor architecture of the node processor,

the communication network and topology of the parallel machine,

the programming language, the process model and the compiler to be supported,

the operating system of different host workstations,

and the tools, monitoring techniques and abstraction levels to be supported.

It is obvious, that the previous goals of the TOPSYS project can not be satisfied completely. But within the design of the tool environment, we strictly have tried to use a hierarchical and modular design concept for supporting flexibility and adaptability. In addition, the extensive use of industry standards simplifies the portability of the TOPSYS environment to various different parallel architectures.

The last goal of TOPSYS concerns the implementation complexity and the interaction with other subsystems of a parallel architecture. The implementation complexity of the tool environment can be decreased by a strong integration with the other subsystems of a parallel computer without influencing the functionality and the performance of these subsystems.

Within the TOPSYS project we take into account the integration with the parallel architecture and the distributed operating system. This project goal will be reached by integration of monitoring and tool support into the parallel machine at different levels of abstraction; at the node processor level, the node architecture level, the topology (communication network) level and the operating system level. This integrated concept leads to a balance between implementation complexity, functionality and performance of all the subsystems of a parallel computer.

1.3 The Ideas and Design Concepts

The TOPSYS project is driven by some basic ideas and design concepts which will be explained in this chapter. It is obvious, that these ideas and design concepts have strong influence on the methodology of using the TOPSYS tools.

The first design decision was the selection of the granularity of parallelism and the corresponding process model. Although we concentrated during the implementation of the tool environment on coarse grain parallelism, we also made the overall conceptual design of TOPSYS appropriate for fine grain parallelism. The reason for the selection of coarse grain parallelism was the concentration on distributed memory machines within the project. Coarse grained programming concepts are at the actual state of technology more suitable and powerful for these types of parallel machines. To express the selected type of parallelism we decided to have a process model which is as close to sequential programming languages as possible. This design concept supports a programmer of sequential programs to upgrade with less efforts to the programming of parallel machines.

The implementation of this process model is the distributed operating system kernel MMK (Multiprocessor Multitasking Kernel) which offers dynamically created and deleted global objects and operations. The basics of the process model are communicating sequential processes where communication is done via mailboxes and synchronization via semaphores. The global objects and operations allow to create parallel programs without having in mind the physical topology of the parallel target computer.

The architecture of the TOPSYS tool environment was influenced mostly by the following design concepts. The first of them results from the experience that all tools for validation and analysis of parallel systems mentioned in the first chapter have one requirement in common; they need runtime information on the dynamic behaviour of the parallel execution. As everybody knows, this runtime information can be gathered by monitors. Therefore, one idea of the TOPSYS project is to implement all tools on top of a common distributed monitoring system [BLT90]. The advantage of this integrated design concept is, that the monitoring system has not to be changed when the functionality of the tools is extended. In addition, all tools will be used with a common monitoring environment. The second idea toward an integrated tool environment concerns the monitoring and instrumentation techniques themselves. Different monitoring techniques can be used for collecting runtime information. The most well known monitoring techniques are simulation, software monitoring by source code instrumentation or object code instrumentation, hardware monitoring and hybrid monitoring. Other problems are the different influences on the parallel execution which are introduced by different instrumentation techniques. A still open question in the parallel world deals with the appropriate monitoring technique. Until now it is very hard to qualify and quantify the retardation and therefore the modification of the parallel execution introduced by different monitoring techniques. This fact gave the motivation for the integration and implementation of different monitoring techniques within the TOPSYS environment.

In the actual phase of the project we deal with object code instrumentation, hardware monitoring and hybrid monitoring. An important feature of our approach is, that the tools of the developement environment are independant of the type of monitor they actually use. Therefore the different monitor types offer the same functionality and can be replaced easily without modifying the tools. This also implies the usage of different monitor types across the several nodes of the parallel computer - a heterogeneous monitoring environment. This feature allows a flexible adaption of the TOPSYS tool environment to all application requirements with respect to program retardation. In addition, measurements of the program slow down when using different monitors within the TOPSYS project allow to qualify and quantify the appropriateness of the different instrumentation techniques. Figure 1.2 shows the integration concept explained in this paragraph.

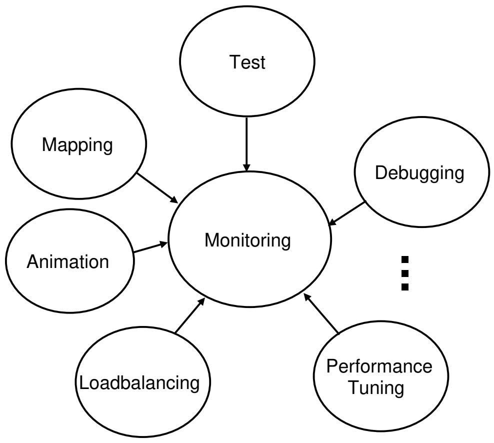


Fig. 1.2: Integration idea

All tools developed within the project allow to analyse the parallel execution at different levels of abstraction depending on the necessary level of detail and granularity. At the highest level (the system level) the programmer looks at the parallel system as a whole; he abstracts from the distinction into several processor nodes. The next view, the node level, focusses on single processor nodes and their interaction. Finally, at the finest level the programmer can analyse parallel constructs (MMK objects, C constructs) and the interaction and communication of them. All objects, no matter whether they are MMK objects or C objects, are referred to at the source code level. This means that the user of the tools refers to all program objects by their names in the source program and the tool environment knows respectively checks the type and the location of these objects.

To support the interactive parallel design cycle illustrated in figure 1.1, all tools and monitors are running concurrently with the parallel application. In contrast to most other parallel development environments known in the field, the TOPSYS tools are used on-line and not in batch mode. Therefore the collection of the monitoring data on the dynamic behaviour of the parallel execution and the representation of these collected monitoring values is done during runtime of the parallel target system. This implies that all tools and monitors represent processes or tasks with respect to the terminology used within the MMK process model.

The last design decision mentioned here should be the support of so called network based host/target environments using industry standards. Most of the available parallel computers today are programmed in host/target environments where program developement (compilation etc.) is carried out on a sequential host workstation and the parallel program is executed on the target machine connected to the host via a local area network. TOPSYS was designed to support these kinds of host/target environments and different parts of the tool environment are distributed between the host and the target. The portability and adaptability of TOPSYS to different network based host/target environments is simplyfied by the use of the following industry standards: within the project we use C and C++ as implementation language, UNIX as host operating system, X-windows as graphics interface, TCP/IP as network protocol and COFF as symbol table format.

2 The Operating System Kernel MMK

This subchapter gives an overview on MMK. We describe the features, design concepts and give an overview on the objects and operations (system calls) offered. In addition, decomposition strategies based on MMK objects and operations are described.

2.1 Features and Design Concepts

There are at the moment two contrary approaches for distributed memory machines in the field. The first and conventional approach offers extensions of existing sequential languages (C, Fortran, Pascal) with parallel constructs. The second approach tries to define new parallel languages based on functional, object oriented and logical programming paradigms. MMK is a compromise between the two approaches, but is closer to the conventional one because of compatibility reasons. In the MMK programming model we tried to enrich the static language extensions of the conventional approach with more dynamic constructs and global operations and objects. The result was a dynamic, transparent process model which allows the programmer to leave the specific architecture out of consideration during program construction. The key features and design concepts of the MMK are:

- The MMK offers a transparent multitasking process model, which means that the programmer can define multiple parallel processes. During program construction the programmer neither has to keep in mind the number of processors nor the mapping from processes to processor nodes.
- When using the MMK programming model, the programmer thinks about his parallel program in an object oriented style. The MMK offers active objects (tasks), communication objects (mailboxes), synchronization objects (semaphores), and storage objects (memory) with the usual operations the programmer can deal with. Object manipulation is only possible via the defined operations.
- All objects and operations are locally and globally available without differences in using them. This means that the programmer makes no differences in using the objects and operations whether the objects are located on the local processor node or on a remote one.
- The MMK offers only a minimum set of objects with their corresponding operations (tasks, mailboxes, semaphores, memory). This design concept of minimality reduces the complexity of implementing dynamic object (process) migration, dynamic load balancing strategies and monitoring support.
- All objects of a parallel program based on MMK can be dynamically created and deleted. This feature implies that the number of objects is only limited by the amount of available memory. This dynamic characteristic of the MMK supports also dynamic load balancing strategies and the implementation of non-numerical applications based on code (task) partitioning.

As already mentioned, one motivation for the MMK was to increase programmer productivity when using parallel computers. This argument also motivated a project in which interactive development tools for parallel applications based on the MMK are developed [Bem88]. These tools are supported by monitoring hooks and interfaces integrated into the MMK. These interfaces and hooks deliver together with software and hardware monitors runtime information for parallel debuggers, performance analyzers, visualization (animation) tools and dynamic load balancers.

In summary, the transparent process model and the global object space of the MMK is illustrated in figure 2.1

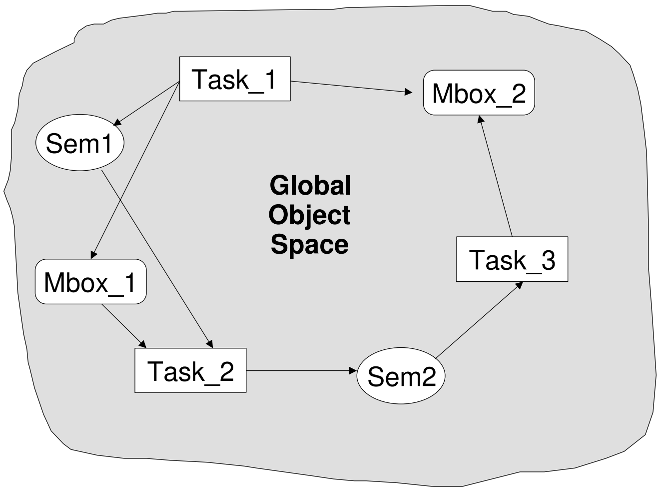


Fig. 2.1: Global object space

2.2 Available Objects and Operations

As already mentioned, the MMK operating system supports three different object types: tasks, mailboxes and semaphores. The programmer can manipulate these objects only via the operations defined on them.

The active parts of an implementation based on MMK are the tasks. In order to meet the multitasking requirements, the tasks on each processor node are scheduled with a priority driven scheduling algorithm. Each task is continuously in one of the following states: terminated, running, ready, waiting, suspended, or waiting suspended. Waiting means that the task is waiting for an access to an object of type mailbox, semaphore, or memory or for timeout. The state changes of the task are initiated by activities of the scheduler or by expilicit system calls (for more details see [Bem87]). Possible system calls are: create task, delete task, suspend task, resume task, set task waiting, set task priority, and get task priority.

The communication between tasks is realized with mailboxes, which allow various forms of interprocess-communication. First of all they support direct communication from one sender to one receiver. Furthermore mailboxes provide means for communication from sender to any receiver (i.e. one receiver from a given set of receivers). So the sender does not necessarily know the receiver of his mail which is useful for certain types of applications. It is possible to define mailboxes with or without buffering, which has to be specified during creation time. With the mailboxes and the timeout mechanism of the mailbox system calls the user can for example implement asynchronous message passing (no-wait-send), synchronous message passing (rendez-vous), buffered and unbuffered message passing. The system calls for mailboxes are: create mailbox, delete mailbox, send message, and receive message.

For the synchronization of tasks the MMK offers counting semaphores with the usual operations defined on them. These semaphores include a timeout mechanism and the semaphore queues are organized in first-in-first-out manner. The following system calls are available for semaphores: create semaphore, delete semaphore, request semaphore, release semaphore.

2.3 The Programming Process

With MMK the process of programming application programs is slightly different from other concurrent programming languages. As the programmer is not forced to keep in mind such things like nodenumbers or maximum number of processing nodes the programming is done on a higher level of abstraction. In fact with MMK the step from the specification of the parallel application to its implementation is very easy.

The first thing the programmer has to do is to devide the application into parts, which can potentially be computed concurrently. These parts form the tasks of the program. If there is any communication necessary between tasks, we have to add some mailboxes or semaphores for synchronization. As a result we get a task graph describing the static structure of our application program.

After having finished the design of the task graph and the programming of the task bodies we can compile the already available source codes.

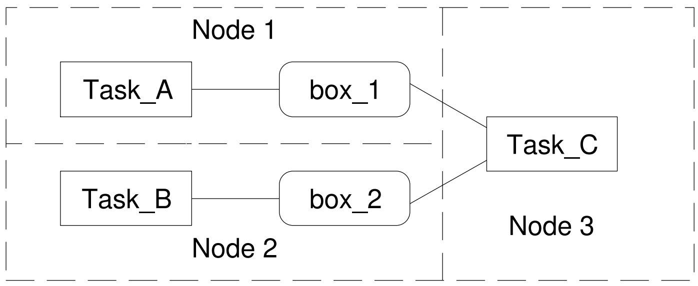
The next step to do is the mapping of the MMK-objects onto the processor nodes of the machine. We will discuss this point in detail in the next paragraph. The information resulting from the mapping is also compiled and linked together with the precompiled sources. The linker yields one big object code file, which now can be loaded into the nodes of the parallel computer. The object code file on every node contains the object code of all tasks which leads to a waste of memory if there is no corresponding task activation on a node. On the other hand, for remote task creations it is absolutely necessary to have all the code on all nodes, otherwise the remote action would fail. So, apart from the above mentioned disadvantage of the possibly inefficient memory usage, our method facilitates linking and loading and shortens the turn-around-time for the program development.

2.4 The Mapping of the Objects

Even if the programming language allows to hide such machine relevant parameters as node numbers, there is a point in the software development cycle, where we have to map the objects of our application program onto the processing nodes of our machine. It is possible to do this mapping automatically and this is also the aim of all efforts done in this area, but, at the moment, there are no adequate solutions for this problem. As a consequence, the user of MMK has to do the mapping by himself, which however brings him some additional facilities of using the machine efficiently.

The mapping is done via a text file which includes the names of all objects to be mapped, the corresponding node numbers and some additional parameters. For task objects we have also to specify the name of the task body, because each task body can have more than one task activation differing only by name.

Figure 2.2 shows an example of a task graph with three tasks and two mailboxes, which are mapped onto three processor nodes. In figure 1.4 we can see the most important part of the mapping file. Lines 1 and 2 define a mailbox on node 1 and node 2, respectively. Line 3 defines a task with task body 'send' which is connected to mailbox 'box_1' and located on node 1. In line 4 we define another sending task, connected to 'box_2' and located on node 2. Finally, in line 5, the receiving task is defined, which is connected to both of the mailboxes and located on node 3.




```
1:  MBOX (box_1 ,..., Node 1)
2:  MBOX (box_2 ,..., Node 2)
3:  TASK (Task_A INIT send ,..., box_1 ,..., Node 1)
4:  TASK (Task_B INIT send ,..., box_2 ,..., Node 2)
5:  TASK (Task_C INIT receive ,..., box_1,box_2 ,..., Node 3)
```

Fig. 2.2: Task graph and corresponding mapping file

The file with the mapping information is transformed into a C-file with global variable declarations and after compilation linked together with the application's object code.

Let us now discuss the two typical parallelization paradigms for parallel computers. Most of the relevant implementations of parallelized algorithms use either data partitioning or task partitioning. We will show that the MMK programming model can deal with both of them.

2.5 Using the MMK for Data Partitioning

One of the most common methods for using parallel computers efficiently is to distribute the load to the processor nodes by dividing the data among them. This method is called data partitioning and results for many applications in a good load balance between the nodes [Fri88]. The process graph is very simple because we have only one task body representing the algorithm. From this task body we make as many task activations as there are nodes in our computer and then map one task onto one node. The partitioning of the data also leads to an inter-task communication but the resulting process graph is regular in most cases: for numerical applications we often use for example ring or torus structures [Ros88].

Example:

Following the so-called replicated-worker principle we implement an algorithm for the solution of a simple partial differential equation system. There is only one task body being responsible for the numerical computation of the resulting matrix. We divide the matrix into stripes and assign each stripe to one activation of the task body. Furthermore we add one mailbox to each pair of tasks computing neighbouring stripes of the matrix because those tasks must exchange data. The resulting process graph is a ring consisting of tasks and mailboxes between them. By using the mapping file we can now adapt the process graph to the available number of processor nodes: we replicate the tasks and the mailboxes, so that the number of tasks equals the number of processors.

So the MMK provides efficient means of using parallel computers for applications based on data partitioning. The programming is very easy and in future the construction of the mapping file can be automized, depending on the number of available processors.

2.6 Using the MMK for Task Partitioning

The second paradigm of using parallel computers is task partitioning. Here the total application is divided into tasks being distinct from each other and solving specialized problems. According to the replicated worker scheme we could call this the specialized worker scheme. In conventional parallel programs, there are two types of specialized workers. First, there are tasks which do not communicate with other tasks in the process graph apart from the fact that starting such a task includes some sort of communication (for example tasks representing whole application programs). Second, there are tasks working in a pipeline-like mode. They receive input-data from a sender, compute some new data and transfer the result to a receiving task, which can be identical to the sending task.

Nowadays, task partitioning is mainly used with distributed systems, e.g. systems consisting of some few coupled workstations, and for the implementation of non-numerical algorithms. With increasing number of processors there are hardly any applications with task partitioning for the simple reason that the number of tasks should exceed the number of processors for using the machine efficiently. This is at least true for tasks with coarse granularity executed most efficiently on today's distributed memory multiprocessors. Task partitioning may become more important with excess parallelism for future multiprocessors at a finer granularity level.

Example:

At our laboratory we implemented one example algorithm with task partitioning: a parallelized parser. Starting with a stack-statemachine we parallelized the algorithm by assigning each state of the stack-statemachine to one process of the process graph. As a consequence each possible state change is represented by a communication channel (= mailbox) between the two states (= tasks). The stack-statemachine works in pipeline-mode, e.g. it is possible (and necessary) to analyze many strings concurrently. The resulting efficiency is not very high, but there are advantages for the programming philosophy. First, with our mapping file, we can arrange the tasks within the machine such that the efficiency is at a maximum. The multitasking facility on each node allows a grouping of tasks to minimize the load imbalance between the nodes. Secondly, software fault-tolerance can be easily achieved by replicating tasks of the process graph with the mapping file on different nodes.

So, also for task partitioning applications the MMK allows an efficient usage of the parallel computer by supporting some kind of static load balancing

As a conclusion we can say that the MMK programming model is suitable for both application programs with data partitioning and programs with task partitioning. But with MMK we can also implement any mixture of the two parallelization types and use the machine efficiently for a wide spectrum of different algorithms.

2.7 Simplified Use of Multiprocessors with MMK

In order to make parallel computers more available for general-purpose applications there will have to be a new programming paradigm which allows a problem oriented programming without taking into account the special qualities of the multiprocessor.

The MMK together with the communication mechanism of these new parallel machines like the iPSC/2 creates a new programming style which has the following advantages:

- First of all, we can use task and data partitioning together. This is an important feature because pure task partitioning is not adequate for the parallel machines as already mentioned and pure data partitioning is no longer required by the parallel machines with the new communication concepts. The mapping file allows a replication of every specialized task so that the number of task activations can be adapted to the amount of data to be computed.

- The multitasking facility reduces the idle time of each processor and even allows multiprogramming to increase the usage of the supercomputer. The replication facility of the mapping file allows to control the granularity of the application program, which is important if we want to use dynamic loadbalancing ([Bai87], [The86]).
- The remote creation facility for tasks allows a dynamic loadbalancing controlled by the programmer as he can activate new tasks on nodes with low load.
- Furthermore the global objects allow a dynamic loadbalancing controlled by the operating system without any repercussion on the source code of the parallel program. This is the biggest advantage of referencing all objects only by name and not by location.

In summary one can say that a heterogeneous software concept being composed of a mixture of task and data partitioning is well suited for programming more heterogeneous machines, for example machines with special nodes for file-I/O, graphic output or sensor input. These special nodes can be efficiently used by assigning specialized tasks to them. So there are several application classes where the advantages of the MMK process model allow an efficient program implementation.

In the field of fault-tolerance we can easily implement software fault-tolerance by replicating processes and the communications between them to different resources. At our laboratory we are now making first measurements to evaluate the costs of such methods.

For numerical algorithms which are mostly using data partitioning we can easily add specialized tasks for the pre- or post-processing of the data (for example I/O-tasks, tasks for graphic). At the moment we are parallelizing a placement algorithm for cells in gate arrays [Kle88].

Also with non-numerical problems, for example backtracking algorithms, we can use MMK efficiently. Here it is often useful to have master tasks producing data to be computed and many slave tasks to do this.

Of course, the comfort of programming with MMK leads to some loss of efficiency as compared to sophisticated mapping schemes for specialized algorithms. But in our opinion it is the consequent step from implementations whose structures are too static to more dynamic ones. It is comparable to the step from machine languages to more problem-oriented programming languages for single processor computers.

3 The TOPSYS Hierarchical Model

In this sub-chapter we give a rough description of the main interfaces. In addition, we give a description of some basic concepts like breakpoint handling.

3.1 The TOPSYS Hierarchical Tool Model

A major result of the research work on the design concepts described in the previous chapter of this paper was the definition of a hierarchical layered model for tool environments like TOPSYS. This layered approach is called the TOPSYS hierarchical tool model and is illustrated in figure 3.1. The figure shows the interfaces defined between the several subsystems of the development environment. The hierarchical model results mainly from the idea to use all tools together with all monitoring techniques, regardless of the implementation of the corresponding

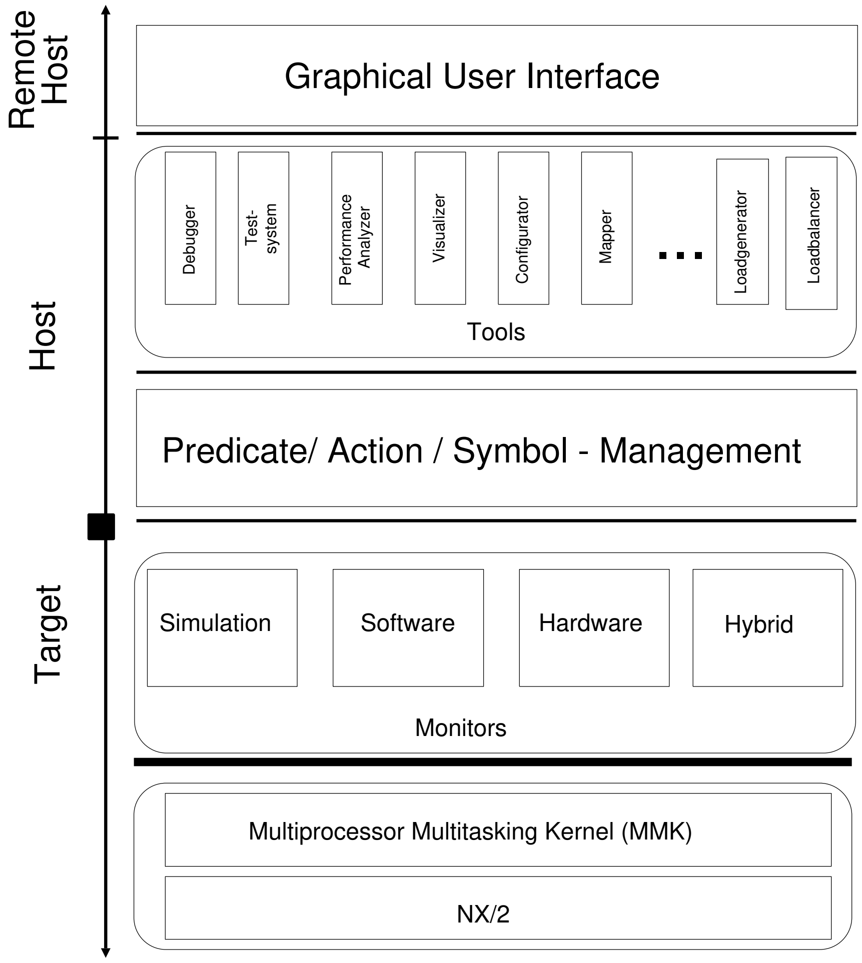


Fig. 3.1: TOPSYS hierarchical tool model

partner. Apart from the definition of some smaller interfaces, the integration idea has lead to the definition of two important interfaces - the monitor interface (UTL, Universal Target Layer) and the tool interface (HLTL, High Level Language Tool Layer). The monitor interface is a command driven interface via which the upper layers of the tool model request the different monitors to deliver runtime information of the processor nodes. The monitors are duplicated and adapted or downloaded to the several processor nodes of the parallel target machine. All monitor types of this distributed monitoring system offer the same functionality to the upper layers of the tool environment and they are therefore replaceable by each other. The monitor interface is based on virtual addresses and internal object identifiers of the MMK kernel, i.e. all objects within the commands of the monitor interface are referred to by their virtual address or object identifier.

The commands of UTL are sent in ASCII format to the nodes. The UTL offers an equivalent functionality as the tool interface, but at a lower abstraction level and restricted to one processor node (local). Therefore, UTL offers nearly the same group of commands as HLTL:

The first group of commands deals with controlling the execution of the application tasks. Commands offered are

G (Go)

S (Step)

The second class of commands is responsible for monitoring program execution. These commands implement the event/action concept of the distributed monitor system of TOPSYS. The commands offered here deal with breakpoints, traces and performance measurements. It is possible to define and clear breakpoints, traces and performance measurements. In addition, commands for inspecting the results of traces and performance measurements are offered. Finally, the above layers can determine via some commands the granularity of performance measurements:

Z (Predicate_Action_Definiton)

B (Breakpoint_Tasks_Definiton)

T (Trace_Definition)

C (Clear_Command)

R (Reset_Tracebuffer)

U (Read_Tracebuffer)

H (Trace_Stop_Position)

F (Set_Time_Unit)

J (Activate_Counter)

K (Read_Counter)

L (Reset_Counter)

Another group of commands is used to look at the state of the processor node. This includes commands which give the list of MMK objects of the node, the state of these MMK objects and the contents of memory cells (bytes, words, etc.).

D (Display_Memory)

Y (Symbol)

V (View_Objects)

I (Inspect_Objects)

The last group of commands is used to modify the state of the processor node. Only one command is within this group for modifying the contents of memory and I/O.

M (Modify)

In contrary to this, the tool interface (HLTL) offers the abstraction level of the source program. All objects of this interface are referred to by their names in the source program, e.g. variable names, task names, etc. The tool interface is a procedural interface. Via this interface all tools can request runtime information about the dynamic behaviour of the parallel execution at the abstraction level of the source program. The advantage of the definition of this interface is the extendability of the tools' functionality without a modification of the lower layers. HLTL offers a set of system calls which allow the tools to control the parallel execution to monitor the execution, to inspect the state and to find out the performance. In addition, some miscellaneous system calls deal with information on the source program. According to this functionality, the system calls of HLTL can be grouped into 6 categories:

- System calls for controlling the execution allow to start tasks or step through tasks (go_s5, sstep_s5, pstep_s5).
- The second class represents the most sophisticated features of TOPSYS, the monitoring of the execution. Here you can set, clear and list breakpoints or traces and look at the trace buffers (dbreak_s5, cbreak_s5, dtrace_s5, ctrace_s5, traceb_s5, ltrace).
- For looking at the state of the program you can list MMK objects, list variables and display the contents of these variables (view_s5, view_nodes_s5, inspect_s5, display_s5).
- For changing the sate of the execution run you can modify the contents of variables (modify_s5).
- For measuring the performance of an MMK program there are system calls for definition of measurements, reading performance values, remove measurements and start performance measurements (pa_define, pa_read, pa_remove, pa_start).
- Additional system calls are for initializing the tool layer and for looking at the symbol table (init_utol, type_s5, view_symbol).

Within the actual phase of the TOPSYS project we have developed several tools with the following functionality based on the tool interface:

- The parallel debugging and test system DETOP for validating parallel programs at the source level.
- A performance analyzer PATOP for perfomance tuning and performance measurement of parallel systems.
- The visualization tool VISTOP for dynamic graphical animation of parallel executions.

The mentioned tools are examples for using the functionality of the monitor and tool interfaces to get insight information on the dynamic behaviour of a running parallel program. It is relatively easy to create new tools with new functionalities based on the runtime information offerd by the tool interface system calls. Examples are more application oriented animation tools, as they are known from batch mode oriented development environments. Another important issue is the usability of all tools at one time based on one program state of the parallel execution. The programmer can very easily (by moving the mouse from one window to another) switch between the different tools. The transformation between the abstraction level of the tool interface (source code) and the abstraction level of the monitor interface (machine code) is done within a central layer of the tool environment. This layer is responsible for the management of specified events, actions and symbol table objects. The services offered by this layer are used by all tools and monitors.

As most of the tools implemented within the TOPSYS project are interactive tools, an important part of the project is the development of appropriate and easy to use user interfaces. All tools of the tool environment have a graphic and menue driven user interface with the same look and feel, i.e. with the same philosophy of usage. For the implementation of this common graphic user interface a specific graphics library on top of X-windows was developed, which is used by all tools of the TOPSYS project. As already mentioned, we have tried to integrate the tool environment into well known network based host/target environments. Therefore the several layers of the tool model run on different machines. The distributed monitor system MONICA and the operating system kernel MMK are implemented on each processor node of the parallel target system. The central transformation layer and the tools are implemented on the host which is at least logically connected to each processor node of the target system. The graphic interfaces can run on each X-window based workstation connected via a local area network to the host computer. This network based approach makes the TOPSYS functionality available to each programmer within the local area network.

3.2 Basic Concepts

In this section of the introduction, we give an overview on some basic concepts used within TOPSYS. We have selected from most system call groups of HLTL one representative, which will be explained. The explanation is done by a rough description of the activities necessary for the selected system call in each layer.

The display functionality should be used as a representative for inspecting and modifying the state of the parallel system. The most interesting case within display is the inspection of a complex data structure (struct, array). A display of such a complex type is subdivided into displays to its primitive types (int, char, etc.) by using the type table. By using symbol table and mapping file (program.global) each access of a primitive type is broken down into display commands of UTL for displaying the contents of the corresponding memory cells. The identification of the relevant local software monitor (processor node) is done via the mapping file. Figure 3.2 gives an overview on display.

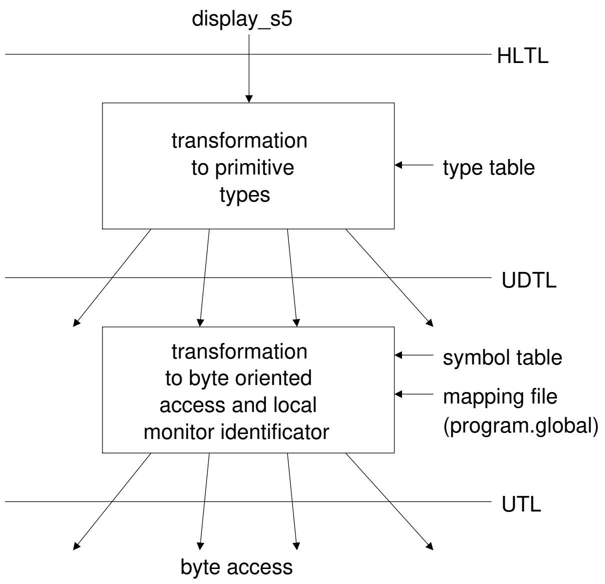


Fig. 3.2: Display

A major feature of TOPSYS is the breakpoint functionality. The breakpoint implementation consists of two parts, the breakpoint definition and the breakpoint message after the breakpoint was detected by the monitors. TOPSYS offers distributed, transparent and global breakpoints. Global means that the breakpoint definition may consist of parts which have to be detected on different processor nodes. The definition is transparent, e.g. the user does not have to know on which processor node the specified parts of a breakpoint are evaluated. The term distributed is used for a real distributed detection (monitoring) of breakpoints.

Breakpoint definition by the tools is done via the HLTL interface procedure dbreak_s5. With the parameters of this system call the type of breakpoint and the action to be initiated upon detecting the breakpoint is specified. Via the symbol table, the symbol references in the predicate specification are translated into virtual adresses. The localizer distributes the several parts of the predicate specification to the local monitor of each processor node. This distribution is done based on the mapping information in xxx.global. The local monitors then evaluate the predicate by monitoring the different events. Combination of events via processor nodes is done using the crosstrigger actions.

If the predicate is fulfilled and the breakpoint happens, two actions are initated. The tasks specified in the breakpoint specification are stopped on each processor node and one local monitor sends an asynchronous breakpoint message to the host layers. In the host layers a separate process is activated by this asynchronous message. This process activates an action within one of the tools, specified as parameter within dbreak_s5. The following figure gives a block diagram of the explained schemes.

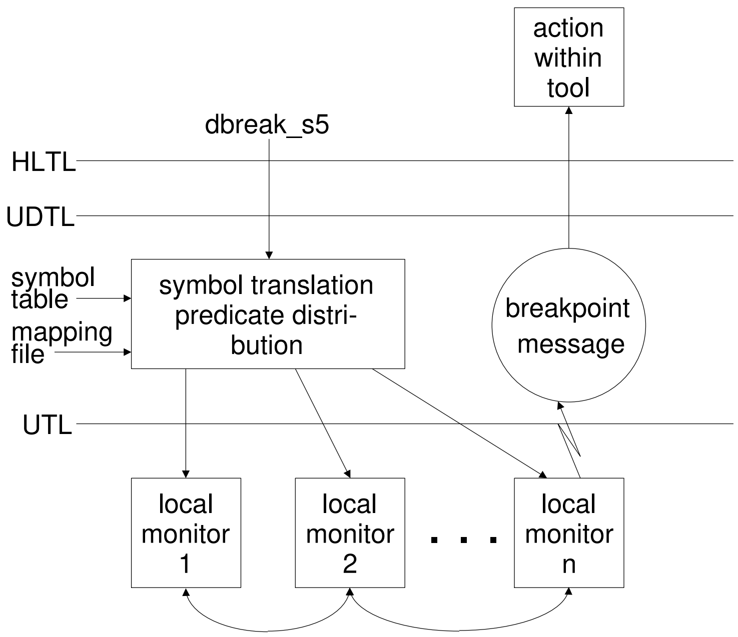


Fig. 3.3: Breakpoint processing

Another group of commands deals with performance measurements. As an example we would explain the principal way of operation of performance measurements. For initiation of a performance measurement, the performance analyzer PATOP calls the HLTL procedure pa_define. The performance measurement to be defined is inserted into an internal database containing all specified measurements. The localizer uses the mapping file and the symbol table to figure out to which local monitor the performance specification has to be sent. After this the specified performance measurement is started via pa_start. To read the performance values from the database the tools can use pa_read. If no appropriate performance values are available, they are collected via the measurement module and the localizer. The principal way of operation of this scheme is demonstrated in the following figure.

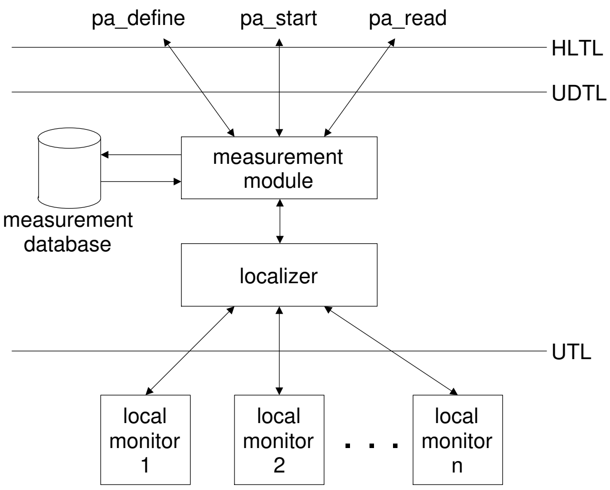


Fig. 3.4: Performance measurement

The last group of functionalities to be explained has to deal only with commands executed on the host. Examples are symbol table oriented commands (type), mapping oriented commands (view_nodes) and miscellaneous functions (help, listing). These commands are implemented on the host by using the several “databases” (symbol table, source, mapping file). The following block diagram illustrates this scheme.

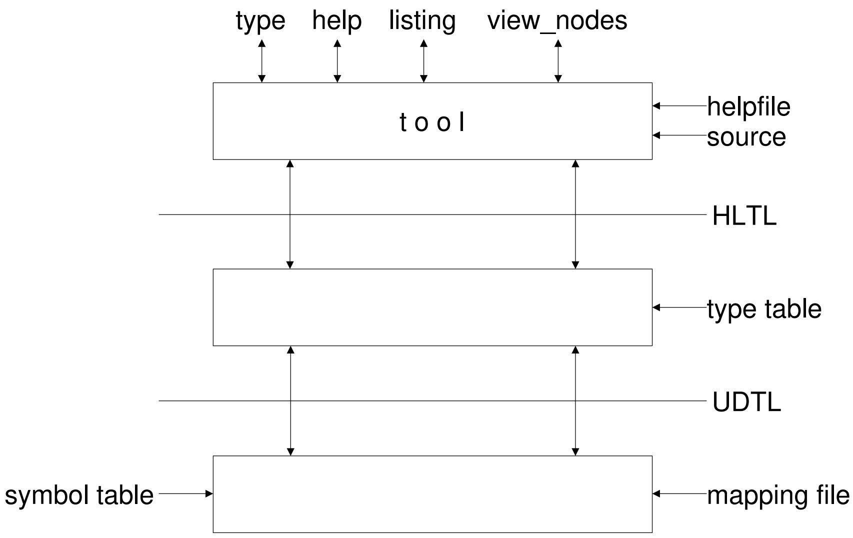


Fig. 3.5: "databases" used on the host

Part 2:

Graphical User Interface (GUI) Objects of the TOPSYS Tools

Peter Braun

This part of the design documents describes the objects of the Graphical User Interface (GUI) of the TOPSYS tools DETOP, PATOP and VISTOP. It consists of two parts, the description of the widgets and the graphical object classes. These classes use the widgets described in the first part to build more complex objects with a nicely to use user interface. The source code for the objects described within this part of the design documents is located in the directory tree below /usr/proj/ipsc2/TOPSYS/HOST/GRAPHIK.

1 Widgets

The GUI of the TOPSYS tools uses the X Window System developed at the Massachusetts Institute of Technology's Project Athena. Our current implementation of the GUI uses version 11 release 4 of the X Window System and is based on the Athena Widget set. Widget is an artificial word based on window gadget. The graphical objects build upon the X Toolkit Intrinsics are called widgets. This part of the TOPSYS design document assumes a basic knowledge of the X Window System, the X Toolkit Intrinsics and the Athena Widget set. See the appropriate documentation on this topic (for example [OREi90],[Cor89]). Especially the concepts of the resources, translations, callback functions, actions and public procedures are necessary.

1.1 Athena Widget Classes Used in the TOPSYS Tools

The Athena widget set is a sample implementation of a widget set using the X Toolkit Intrinsics. The description of the standard Athena Widgets stems partly from the X Athena Widget standard documentation. Here only the functionality of the widget classes is described, however not the interface (e.g. resources, actions).

The TOPSYS tools use the following widget classes:

- labelWidgetClass
  A Label widget is a text string or bitmap displayed within a rectangular region of the screen (e.g. a title)
- commandWidgetClass
  The Command widget is an area, often rectangular, that contains a text label or bitmap image. Those selectable areas are often referred to as “buttons”. When the pointer cursor is on a button, it becomes highlighted by drawing a rectangle around its perimeter. This highlighting indicates that the button is ready for selection. When pointer button 1 is pressed, the Command widget indicates that it has been selected by reversing its foreground and background colors. When the button is released, the Command widget's notify action will be invoked, calling all functions on its callback list.
- textWidgetClass
  The Text widget provides text editing facilities for longer texts.
- listWidgetClass
  The List widget contains a list of strings formatted into rows and columns. When one of the strings is selected, it is highlighted, and the List widget's notify action is invoked, calling all routines on its callback list. Only one string may be selected at a time.

- dialogWidgetClass

  The Dialog widget implements a commonly used interaction semantic to prompt for auxiliary input from a user. For example, you can use a Dialog widget when an application requires a small piece of information, such as a filename, from the user. A Dialog widget, which is simply a special case of the Form widget, provides a convenient way to create a preconfigured form. The typical Dialog widget contains three areas. The first line contains a description of the function of the Dialog widget, for example, the string “Filename:”. The second line contains an area into which the user types input. The third line can contain buttons that let the user confirm or cancel the Dialog input. Any of these areas may be omitted by the application.
- viewportWidgetClass

  The Viewport widget consists of a frame window, one or two Scrollbars, and an inner window. The size of the frame window is determined by the viewing size of the data that is to be displayed and the dimensions to which the Viewport is created. The inner window is the full size of the data that is to be displayed and is clipped by the frame window. The Viewport widget controls the scrolling of the data directly. No application callbacks are required for scrolling. When the geometry of the frame window is equal in size to the inner window, or when the data does not require scrolling, the Viewport widget automatically removes any scrollbars. The forceBars option causes the Viewport widget to display all scrollbars permanently.
- formWidgetClass

  The Form widget can contain an arbitrary number of children or subwidgets. The Form provides geometry management for its children, which allows individual control of the position of each child. Any combination of children can be added to a Form. The initial positions of the children may be computed relative to the positions of other children. When the Form is resized, it computes new positions and sizes for its children. This computation is based upon information provided when a child is added to the Form. The default width of the Form is the minimum width needed to enclose the children after computing their initial layout, with a margin of default distance at the right and bottom edges. If a width and height is assigned to the Form that is too small for the layout, the children will be clipped by the right and bottom edges of the Form.
- popupShellWidgetClass

  Pop-up widgets are used to create windows that are outside of the window hierarchy defined by the widget tree. Each pop-up child has a window that is a descendant of the root window so that the pop-up window is not clipped by the pop-up widget’s parent window. Therefore, pop-ups are created and attached differently to their widget parent than from normal widget children. A parent of a pop-up widget does not actively manage its pop-up children; in fact, it usually never notices them or operates upon them.

All pop-up widget classes are responsible for communicating with the X window manager and, therefore, are subclasses of a Pop-Up Shell.

- toplevelShellWidgetClass
  Shell widgets hold an application's top-level widgets to allow them to communicate with the window manager.

1.2 Additional Widgets of the TOPSYS tools

The Athena Widget Set is a usefull basis for the GUI of the TOPSYS tools, however they require additional functionality. Therefore a number of additional widgets had to be created.

1.2.1 barWidgetClass

The Bar widget is a rectangle which holds a real value between 0 and 1 and presents it as a bar graph. It can be used both as an input and an input/output widget. The bar can be presented vertical and horizontal.

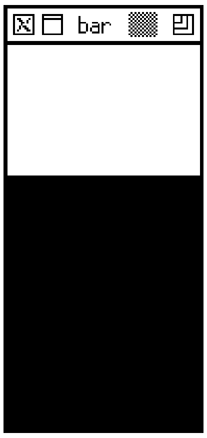


Fig. 1.1: Bar widget filled 66%

The widget can work in output mode and in input/output mode which is determined by the resource XtNchangeable.

output mode (XtNchangeable = False)

The method XtBalken_setzen(Widget w, float percent) draws a black bar of length percent.

input/output mode (XtNchangeable = True) output:

the output is identical to the output in the output only mode

input:

On entering the bar area, the mouse cursor changes its form (double arrow). Pressing the left mouse button and moving while this button is pressed causes the callback procedure XtNmoveProc to be executed. Releasing the mouse button calls the callback function XtNchangedProc and the bar gets updated.

Creation of an instance of a Bar widget loads the following resources from the resource database:

Name	Type	Default	Description
XtNbackground	Pixel	White	background colour
XtNchangeable	Boolean	FALSE	mode (input/output)
XtNchangedproc	XtCallbackList	NULL	callback procedure moving the mouse
XtNforeground	Pixel	Black	foreground colour
XtNhCursor	Cursor	XC_sb_up_arrow	cursor for horiz. bar
XtNheight	int	length or thickness	height of the bar
XtNlength	int	100	length of the bar
XtNmoveProc	XtCallbackList	NULL	callback procedure release of the mouse
XtNorientation	XtOrientation	XtorientVertical	orientation of the bar
XtNshow	float	0.0	shown value
XtNthickness	int	14	thickness of the bar
XtNvCursor	Cursor	XC_sb_right_arrow	cursor for vert. bar
XtNwidth	int	thickness or length	width of the bar

The barWidgetClass uses the following default translations:

```
<Btn1Down>:       NotifyMove()
<Btn1Motion>:     NotifyMove()
<BtnUp>:          NotifyChange()
```

NotifyMove calls the XtNmoveProc callback list, NotifyChange calls the XtNchangedproc callback list and updates the bar.

Each callback procedure gets a parameter of the type _position.

```
typedef struct  {
        int        pixel;
        float      proz;
        } _position;
```

i.e. a callback procedure to be called by a bar widget must have the interface:

```
void my_callback (w, closure, position)
    Widget      w;
    caddr_t     closure;
```

_position *position;

An instance of the Bar widget class can be created by XtCreateWidget. The class variable must be set to barWidgetClass.

1.2.2 buttonWidgetClass

The Button widget is an extended Command widget. The only difference is the interface of the callback function that is called, when a mouse button is pressed within the window. This callback function has an additional argument which contains the number of the mouse button.

1.2.3 dialogWidgetClass (DialogS)

This widget class is very similar to the Dialog widget of the Athena widget set. The only additional functionality is the possibility to change the default user input of the dialog widget with function

XawDialogSetValueString(Widget dialog,char *string) after creation.

1.2.4 iconWidgetClass

The iconWidgetClass cooperates with the slobjectWidgetClass (1.2.5). Both classes together can be used to build objects that can be iconified and deiconified which is usually only possible for toplevel windows. You have to assign another widget to the Icon widget (this will be most times an Slobject widget). The image of an Icon widget (resource XtNbild) is a Pixmap. If you deiconify an Icon widget, the associated widget (resource XtNassoc) will be shown on the screen. You can deiconify an Icon widget with the left mouse button or by calling the public function

Boolean XtDeiconify(/* Widget w */).

The return value tells you, if the call was successful (TRUE = o.k.).

Resources of the Icon widget:

Name	Type	Default	Description
XtNassoc	Widget	NULL	associated Widget
XtNbild	Pixmap	NULL	icon image
XtNdeic_callback	XtCallbacklist	NULL	deIconify-Callback
XtNmiddleb_call	XtCallbacklist	NULL	middle button Callback
XtNrightb_call	XtCallbacklist	NULL	right button Callback
XtNfont	XFontStruct *	6x10	font for the label
XtNforeground	Pixel	White	foreground colour
XtNname	char *	BLANK	name of the icon

An instance of the iconWidgetClass can be created with the function XtCreateWidget.

1.2.5 slobjectWidgetClass

The slobjectWidgetClass cooperates with the icon widget class(1.2.3). An icon can be assigned to an Slobject widget (resource XtNicon). You can iconify an Slobject widget with the left mouse button or by calling the public function

Boolean XtIconify(/* Widget w */);

The return value is TRUE, if the conversion was successful.

The Slobject widget is a subclass of the Form widget. The only difference between the original Form widget and the Slobject widget is the additional icon and the callback lists assigned to the mouse button presses.

The Slobject widget has the following resources in addition to the form widget:

Fig. 1.2: Slobject widget and the assigned Icon widget

Name	Type	Default	Description
XtNicon	Widget	NULL	associated Widget
XtNic_callback	Pixmap	NULL	iconify widget
XtNmiddleb_call	XtCallbackList	NULL	middle button Callback
XtNrightb_call	XtCallbackList	NULL	right button Callback

An instance of the slobjectWidgetClass can be created by the function XtCreateWidget.

1.2.6 messbalkenWidgetClass

The Messbalken (measure bar) widget is a window that enables digital and analogous input and output. It consists of four elements:

- a headline (Label wigdet)
- the digital output (Label Widget)
- the analogous input output area (Bar widget)
- the scale (Skala Widget)

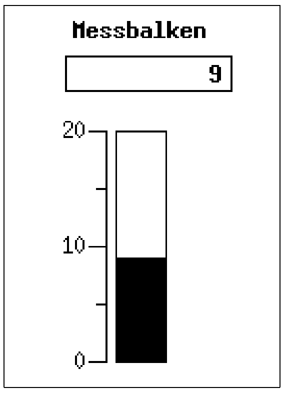


Fig. 1.3: Messbalken widget with linear scale

In figure 1.3 you can see a Messbalken widget. The scale can be linear or logarithmic (resource XtNlog). The bar can be horizontal or vertical (resource XtNorientation). In oder to use the Messbalken widget as an input device, you have to select XtNchangeable = TRUE. Then you can modify the Bar widget by moving the mouse pointer, while the left mouse button is pressed. After you release the mouse button, the bar and the digital value will be updated. You can also change the value of the Messbalken widget, by using the public functions

void XtZeigeWert (/* Widget w, long value */) or
void XtZeigeDouble (/* Widget w, double value */).

The Messbalken widget uses the following resources:

Name	Type	Default	Description
XtNakt_wert	XtCallbackList	NULL	mouse position
XtNchangeable	Boolean	True	switch in/output mode
XtNlabel	char *	“Messbalken”	label
XtNlaenge	int	120	length in Pixel
XtNlog	Boolean	False	linear/logarithmic
XtNmax	long	120	max. Value of the scale
XtNmin	long	0	min. Value of the scale
XtNorientation	XtOrientation	XtorientHorizontal	horizontal/vertical
XtNwert	long	0	shown value

An instance of the messbalkenWidgetClass can be created by the function XtCreateWidget.

1.2.7 mlistWidgetClass

The Mlist (Multi list) widget is based on the List widget of the Athena widget set. The Mlist widget, however, allows you to select more than one entry out of a list. The class Mlist exports analogous functions as List (XtMListChange, XtMListUnhighlight, XtMListHighlight and XtMListShowCurrent). XtMListShowCurrent however returns a list of all selected items.

```
XtMListReturnList * XtMListShowCurrent (/* Widget w */)
typedef struct _XtMListReturnList {
        String          string;
        int             index;
        struct _XtMListReturnList *next;
        } XtMListReturnList;
```

Note that in X11 R4 all the function names of the List widget start with Xaw instead of Xt.

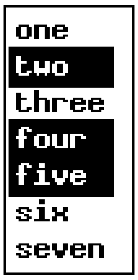


Fig. 1.4: Mlist widget with 3 items selected

1.2.8 otoggleWidgetClass

The Otoggle widget is rectangular area which represents yes/no decision. Usually an instance of an Otoggle Widget is used together with an instance of a label widget which determines the purpose of the toggle. If an item is selected this is indicated by a little hook in the Otoggle widget If the item is not selected, the Otoggle widget is empty. The Otoggle widget can be used to build ’radio buttons’, i.e. out of a group of Otoggle widgets is always at most one selected. In X11 R4 there is an Athena widget called Toggle widget that incorporates this functionality.

The Otoggle widget uses the following resources:

Name	Type	Default	Description
setcallback	XtCallbackList	NULL	otoggle is set
unsetcallback	XtCallbackList	NULL	otoggle is unset
background	Background	White	background colour
cursor	Cursor	opendot	cursor form
foreground	Foreground	Black	foreground colour
lineThickness	Thickness	1	thickness of the hook
sensitive	Sensitive	True	sensitive state
set	Set	False	initial value
width	Dimension	20	x dimension
height	Dimension	20	y dimension
x	Position	0	x position
y	Position	0	y position

You can create an Otoggle widget by the funciton XtCreateWidget with the class variable otoggleWidgetClass.

1.2.9 loadWidgetClass

The Load widget can be used to display load values. The load history is shown as a graph similar to the load values of the xload tool which is part of the X environment. The load widget is used as part of the histo class (figure 2.2). You can display load values with the public function

```
void draw_load(/*Widget w,int start_time,end_time,value */).
```

start_time and end_time specify the start and end time of a measure intervall in milliseconds. This intervall is scaled by the resource timeperpixel. The parameter value is the load value of the measure. It is divided by (end_time - start_time) to get the average load for this intervall. The resource scale is always adjusted that the maximum average load value is between scale/2 and scale. The load values will be shown until the load window is filled. Then the left half part of the window is removed and the right half of it slides to the left (like xload). When you specify a focus_time different to -1, the focused intervall is displayed.

Every time when the load widget is updated, the focusProc callback list is called with a parameter of type *Focus_integral.

```
typedef struct {
        int         integral;    /* average load value of the focus intervall */
        int         start;       /* start time of focus intervall */
        int         interval;    /* length of focus interval   */
        int         skala;       /* load scale         */
        Boolean     focused;     /* FALSE if no focus value is returned */
    } Focus_integral;
```

The Load widget uses the following resources:

Name	Type	Default	Description
width	Dimension	120	width
height	Dimension	120	height
scale	int	1	max. value
minscale	int	1	min. value for scale
label	String	“”	name
foreground	Pixel	White	foreground
background	Pixel	black	background
font	XFontStruct *	fixed	font of the label
focusProc	XtCallbackList	NULL	change of focus
reverseVideo	Boolean	FALSE	
timeperpixel	int	5	resolution
focus_time	int	0	start time of focus intervall
focus_length	int	0	length of focus intervall
time_sum	int	0	measure period

You can create a Load widget by the funciton XtCreateWidget with the class variable loadWidgetClass.

1.2.10 sldeskWidgetClass

The Sldesk widget is simple a container for other widgets, i.e it is a subclass of the composite widget. Instead of the standard Athena widgets it does no layout on its children. The sldeskWidgetClass has a callback list assigned to expose events. The Sldesk widget is used by VISTOP as the animation panel.

The Sldesk widget uses the following resources:

Name	Type	Default	Description
foreground	Pixel	black	
background	Pixel	white	
exposeCallback	XtCallbackList	NULL	expose callback list

You can create a Sldesk widget by the funciton XtCreateWidget with the class variable sldeskWidgetClass.

2 C++ Graphical Interface

The X Window System architecture is based on a object oriented programming paradigm (Class structure, inheritence ...). However it is not possible to use the widgets from the Athena widget set from C++ without modification. Their interfaces are written in the Kernighan & Ritchie C style and not in the ANSI C style required by C++. In order to provide a C++ interface for the graphical objects of the TOPSYS tools we used two different approaches:

1) We used the widgets described in chapter 1 to build more complex and comfortable objects. The graphic routines for this approach were implemented in C and supplied with a C++ interface. The TOPSYS tools are using these objects only via their C++ interface.

2) We added C++ header files to all the Athena widgets. This task is straightforward. You only have to add function prototypes to the widget convenience functions. Our own widgets interface files have both Kernighan & Ritchie C style and ANSI C interfaces included. The appropriate style can be selected by C preprocessor instruction (#define C++).

This part of the document describes the graphical object classes and their interfaces which were built using the first method. These are the classes Bargraph, GenOut, Histo, InOut, Menu and Message. The description is in the syntax of a C++ class definition, however only public methods are described, but not the private data structures and private methods.

Note that all parameters in the following methods which characterize positions or sizes of objects require values in pixels rather than in number of characters or size in mm.

2.1 The Bargraph Class

The Bargraph class can display at the same time measurements of different objects. The values are shown analog (as bar) and digital (above each bar). Figure 2.1 shows a Bargraph window which is used by PATOP. The Bargraph class uses several different widget classes.

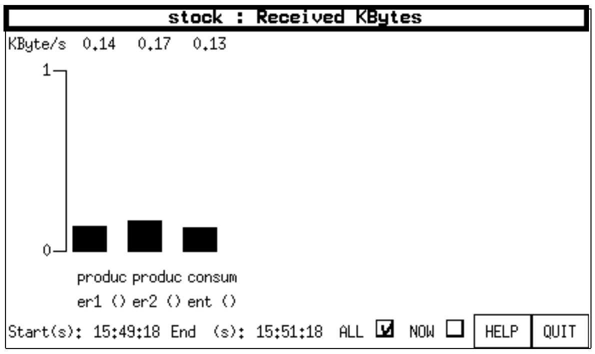


Fig. 2.1: Bargraph window how it is used by PATOP

The class Bargraph has the following public functions:

FUNCTION: Bargraph::Bargraph(struct timeb* time, char* units, char* title,
void (*quitproc)(caddr_t),caddr_t quitdata,
XtCallbackProc(*helpproc)(Widget,caddr_t,caddr_t),
caddr_t helpdata)

Parameter:	time	start time of the measure
	units	unit of the measure
	title	title of the measure
	quitproc	pointer to the function which is executed, when the "QUIT" buttons is pressed
	quitdata	pointer which is passed to the function "quitproc" as argument when it executes
	helpproc	pointer to the function which is executed, when the "HELP" buttons is pressed
	helpdata	pointer which is passed to the function "helpproc" as argument when it executes

Description: This is the Constructor of the class Bargraph. It creates a raw Bargraph window and initializes it. However it does not include the bars.

The Bargraph window consists of the following parts:

titlewidget:	title of the measure (Label widget)
unitwidget:	units of the measure (Label widget)
skalawidget:	scale for the bars (Skala widget) the skalawidget gets a call back function that updates the max value of the scale
quitbutton:	closes the Bargraph window and calls the user definable quit procedure quitproc.
helpbutton:	calls the user definable help procedure helpproc
alltoggle, nowtoggle:	each toggle consists of a Label wiget and a Otoggle widget in a form widget. Always exactly one of the toggles is on, i.e. they act as radio buttons. They also have callback functions assigned to update the bars. If the nowtoggle is on the bars show only the last entered value. All means that the bars show the average over all values.

Finally the bar window has another four label widgets. They show the start and end time of the measure intervall.

FUNCTION: void Bargraph :: Add(char* name)

Parameter: name — name of the new bar

Description: Add is the method that adds another bar in the bar diagram at the right side. Each bar consists of three parts:

- valuewidget: this is a Label widget that can hold the digital value of the bar
- barwidget: this is a bar widget that shows the analogue value of the measure.
- namewidget: it contains the name for the bar. If this name is too long it is split up in two parts. This part consists of two label widgets.

FUNCTION void Bargraph ::Set(char* name, Pa_value value, Boolean updtime)

Parameter:

- name — name of the bar whose values is to be changed
- value — list of measure values
- updtime — bool value that determines, if the measure intervall of value has to be added to the whole measure time.

Description: This method doesn't add new widgets like former two. It adds new values to the bar name of the Bargraph window and updates it according to the update mode ('all' or 'now').

The updtime parameter determines if this set call shall update the intervall for the whole measure time. You will probably get one measure value list for each bar, where all value lists refer to the same time intervall. So you only want to update your total measure time intervall once for one measure intervall though you have more than one set call.

2.2 The Histo Class

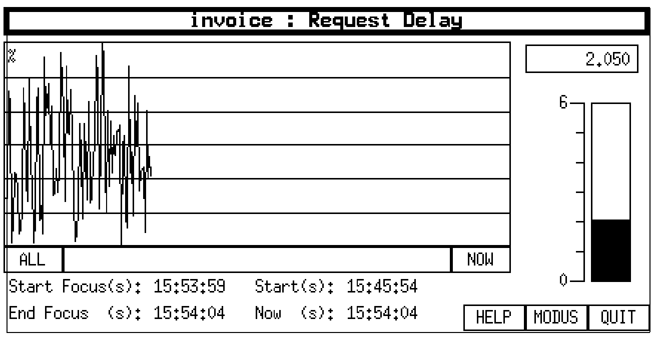


Fig. 2.2: Histo window as it is used by PATOP

The Histo class can be used to build curve diagrams (see figure 2.2 for an example). Such a curve diagram shows how one value evolves over the time. In addition the last measured value or the average value is displayed as a bar (analogous) and digital.

The Histo Class has the following public functions:

FUNCTION: Histo::Histo(struct timeb* time, char* units,
char* title, void (*quitproc)(caddr_t),
caddr_t quitdata,
XtCallbackProc(*helpproc)(Widget,caddr_t,caddr_t),
caddr_t helpdata,int pixeltime)

Parameter:

time	start time of the measure
units	unit of the measure
title	title of the measure
quitproc	pointer to the function which is executed, when the “QUIT” buttons is pressed
quitdata	pointer which is passed to the function “quitproc” as argument when it executes
helpproc	pointer to the function which is executed, when the “HELP” buttons is pressed
helpdata	pointer which is passed to the function “helpproc” as argument when it executes
pixeltime	time in ms that takes one pixel in the curve

Description: This is the Constructor of the class Histo. It creates and initializes a Histo window for a curve diagram.

The Histo window consists of the following parts:

titlewidget:	title of the measure (Label widget)
loadwidget:	loadwidget is a Load widget. It has assigned the callback function FocusProc which executes whenever loadwidget get updated. FocusProc makes an update of the Messbalken widget and the time entries.
allbutton:	Command widget with the label ’All’. It has assigned a callback function which switches the Load widget in the ’all’ mode (focus is the whole measure intervall). The update is delayed until FocusProc executes the next time.
nowbutton:	Command widget with the label ’Now’ It has assigned a callback function which switches the Load widget in the “now” mode (no focus intervall). The update is delayed until FocusProc executes the next time.

scrollbar:	Scrollbar widget which can be used to specify a focus intervall. The Scrollbar widget has two callback functions assigned. One is called when the position of the scrollbar is changed, the other function is called when the width of the scrollbar is changed. In each case the load widget will be updated according to the changes.
quitbutton :	closes the Bargraph window and calls the user definable quit procedure quitproc.
changebutton:	this is a command widget with label 'MODUS' and an assigned callback procedure which switches between two display modes. The default is shown in figure 2.2. In the second mode only the area of the curve graph is shown.
helpbutton:	calls the user definable help procedure helpproc
messbalken:	the Messbalken widget shows the curve displayed in the Load widget as a bar and a digital value. The value is computed according to the selected focus. It is the last measured value if no focus is selected (i.e. the 'NOW' button was pressed) or the average value over the focus intervall
skalawidget:	scale for the bars (Skala widget) the skalawidget gets a call back function that updates the max value of the scale

Finally the Histo window has another eight label widgets. They show the start and end time of the measure intervall and start and end time of the focus intervall.

FUNCTION:	void Histo::Set(Pa_value pa_value)
Parameter:	pa_value is a pointer to a list of measure values
Description:	The method Histo::Set adds the values of pa_value to the curve diagram.

2.3 The InOut Class

The InOut Class provides a comfortable and powerful C++ graphical input/output interface. It allows you to combine an arbitrary number of Button, Dialog, Text, Toggle, List and Mlist widgets to form an input/output window (see figure 2.3 for an example). The input area (left part of the window) lets you specify the kind of information you are interested in, the output area (right part of the window) shows the results of your requests in textual form. The layout algorithm for the different components of the input area is as following: Widgets of the same class are placed from the left to the right within the same "line" (class refers to Button, Dialog and so on). This order can be changed by the methods newline and separator. They cause the next object to be placed in the next row. The class InOut really consists of three classes. The class InOutMethods serves as superclass of the classes In and InOut. The classes In and InOut provide nearly the same functionality to the user, however the In class has no output window assigned.

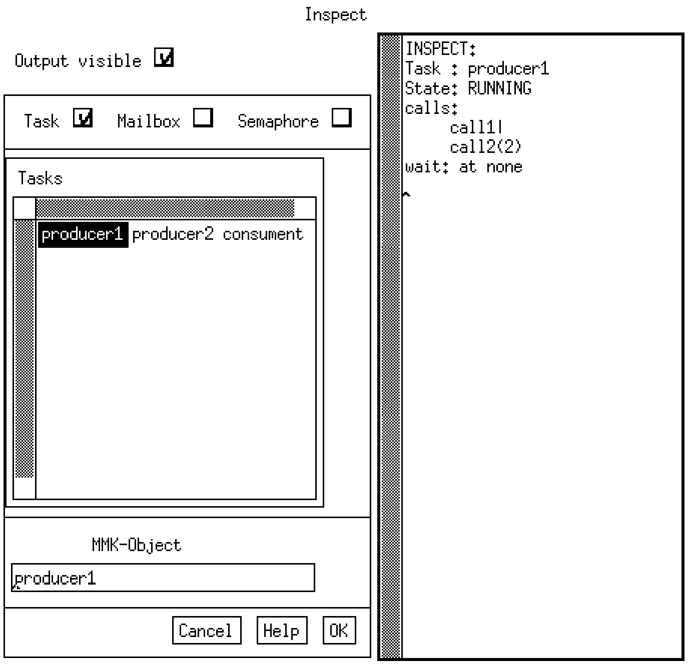


Fig 2.3: InOut window

Description of the methods of the InOut Class:

FUNCTION:	In::In(char* title, Widget parent, char* font, int xpos, int ypos);	
Parameter:	title:	title of the input window
	parent:	parent widget of the input window
	font:	title font
	xpos:	x position of the window
	ypos:	y position of the window
Description:	The method In::In is the constructor for an input window. It serves as initialization routine and must be executed before one of the following functions can be used. It first creates a Popup shell widget and a form widget which is used as a container for all other widgets to be added. Finally a Label widget with the title of the In window is added.	

FUNCTION: In::~In();
Parameter: no parameter
Description: The method In::~In is the destructor for an input window.

FUNCTION InOut::InOut(char* title, Widget parent, char* font, int xpos, int ypos);
Parameter:
- title: title of the input/output window
- parent: parent widget of the input/output window
- font: title font
- xpos: x position of the window
- ypos: y position of the window

Description: The method InOut::InOut is the constructor for an input/output window. It serves as initialization routine and must be executed before one of the following functions can be used. It first creates a Popup shell widget and a form widget which is used as a container for all other widgets to be added. Then a Label widget with the title of the InOut window is added. So far the In and Inout constructor are identical. The only difference between both is an additional output window and a toggle button which serves to show respectively hide the output window.

FUNCTION: InOut::~InOut();
Parameter: no parameter
Description: The method InOut::~InOut is the destructor for an input/output window.

All methods of the class InOutMethods are inherited from both the In and the InOut class.

FUNCTION: void InOutMethods::Popup(int xpos, int ypos);
Parameter:
- xpos x position of the upper left corner
- ypos: y position of the upper left corner

Description: Pops the receiver window up at screen position xpos,ypos

FUNCTION: void InOutMethods::Popdown();
Parameter: no parameters
Description: Pops the receiver window down

FUNCTION: void InOutMethods::Newline();
Parameter: no parameters
Description: This method changes the layout rule for the next widget to be added. The next widget will not be placed in the same line. It will be placed at the next line.

FUNCTION:	void InOutMethods::Seperator();	
Parameter:	no parameters	
Description:	This method changes the layout rule for the next widget to be added. The next widget will not be placed in the same line. It will be placed at the next line. Below the last added widget a separation line is included.	

FUNCTION:

```
void InOutMethods::AddButtonTextLeft
                    (char* name, int dist, XtCallbackProc cb, caddr_t cb_data);
void InOutMethods::AddButtonPixLeft
                    (char* name, int dist, XtCallbackProc cb, caddr_t cb_data);
void InOutMethods::AddButtonTextRight
                    (char* name, int dist, XtCallbackProc cb, caddr_t cb_data);
void InOutMethods::AddButtonPixRight
                    (char* name, int dist, XtCallbackProc cb,caddr_t cb_data);
```

Parameter:	name:	Label (or name of the Pixmap file) of the button
	dist:	Distance of the predecessor or border
	cb:	Callback procedure to be called
	data:	call data for the callback procedure
Description:	This method adds a Label or Pixmap button to the receiver of the message. The button is either added from the left to the right (Addbutton*Left) or from the right to the left (Addbutton*Right). If a newline or separator call preceds, the button is placed in the next row.	

FUNCTION:

```
void InOutMethods::AddDialog( char* label, char* init, int width, int dist);
void InOutMethods::AddDialogScrollbar( char* label,
                    char* init, int width, int dist);
```

Parameter:	name:	name of the dialog widget
	init:	initial value of the text area
	width:	width of the dialog box
	dist:	distance of the previous window
Function:	In the current implementation AddDialogScrollbar has the same functionality as AddDialog. It creates a window consisting of a label and a one line text box below where text can be entered.	

FUNCTION:

```
void InOutMethods::GetDialogString( char* name, char** value);
void InOutMethods::SetDialogString( char* name , char* value);
void InOutMethods::SetDialogLabel( char* name, char* value);
```

Parameter:	name:	name of the dialog widget
	value:	holds the string to be read or written

Description: This methods change the values of the label (SetDialogLabel) respectively the dialog box (SetDialogString) or read the value of the dialog box (GetDialogString).

FUNCTION: void InOutMethods::AddLabelText(char* name, int dist);
void InOutMethods::AddLabelPix(char* name, int dist);

Parameter:
- name: name of the Label widget or pixmap to be added
- value: holds the string to be read or written

Description: These methods add either a Text Label widget (AddLabelText) or a Pixmap (AddLabelPixmap).

FUNCTION: void InOutMethods::AddTextString(char* text, int dist);
void InOutMethods::AddTextString(char* text, int dist, int width,int height);
void InOutMethods::AddTextFile(char* text, int dist, int width, int height);

Parameter:
- text: text or name of the file to be included
- dist: distance from the previous window
- width: width of the text area
- height: height of the text area

Description: This methods add either a Text widget containing either a string (AddTextString) or the contents of a file (AddTextFile). AddTextString is an overloaded function. If width and height are not specified, the size of the Text widget depends on the string to be displayed. AddTextFile produces a Text widget with a scrollbar. The displayed text can not be changed.

FUNCTION:
```
void InOutMethods::AddToggleText(char* name, int dist,
                          XtCallbackProc setfunc, caddr_t setdata,
                          XtCallbackProc unsetfunc, caddr_t unsetdata);
void InOutMethods::AddTogglePix( char* name, int dist,
                          XtCallbackProc setfunc, caddr_t setdata,
                          XtCallbackProc unsetfunc, caddr_t unsetdata);
```

Parameter:
- name: name of the toggle or file to be included
- dist: distance from the previous window
- setfunc: Callback if Toggle is set
- setdata: Data structure which is passed to the set function
- unsetfunc: Callback if Toggle is unset
- unsetdata: Data structure which is passed to the unset function

Description: These methods add either a toggle widget together with a textlabel or a together with a pixmap. The textlabel or the pixmap is located to the left side of the toggle widget. The toggle has assigned two callback functions. One is

	executed when the toggle is set (this is indicated by an arrow in the toggle rectangle) the other is executed, when the toggle is unset.	
FUNCTION:	void InOutMethods::SetToggle(char* name); void InOutMethods::UnsetToggle(char* name);	
Parameter:	name:	name of the toggle
Description:	These methods set or unset a toggle widget. They can be used to implement radio buttons. These methods don't call the associated callback functions.	
FUNCTION:	void InOutMethods::SetToggleLabel(char* name , char* label);	
Parameter:	name:	name of the toggle
	label:	new label for the toggle
Description:	This method changes the label of the toggle window to the new value 'name'.	
FUNCTION:	void InOutMethods::ResetToggle(Widget w, caddr_t closure, caddr_t callData);	
Parameter:	w:	toggle widget to reset
	closure:	callback function
	callData:	Parameter of the Callback function
Description:	This method resets the toggle widget.	
FUNCTION:	void InOutMethods::AddList(char* name, String* List, int col_num, int distance, int width, int height)	
Parameter:	name:	name of the List widget
	List:	List of the entries
	col_num:	number of columns in the list
	dist:	distance to the previous widget
	width:	width of the widget
	height:	height of the widget
Description:	This method adds a list widget to the InOut window.	
FUNCTION:	void InOutMethods:: GetList(char* name, char** value) void InOutMethods:: GetList(char* name, char*** List, char** value, int* index)	
Parameter:	name:	name of the List widget
	List:	pointer to the whole list
	value:	selected item in the list

	index:	index of the entry within the list
Description:	This is an overloaded function. The first version get the selected item from a list widget as a string. The second version also receives a pointer to the list as a whole and the number of the selected entry in the list.	

FUNCTION: void InOutMethods:: SetList(char* name, String* List,
char* newlabel,
XtCallbackProc callback,
caddr_t callData)

Parameter:	name:	name of the List widget
	List:	pointer to the whole list to add
	newlabel:	set the label of the list window to 'newlabel'
	callback:	associated callback function
	calldata:	data for callback
Description:	This method replaces the attributes of a list structure. The parameters List, newlabel and callback may have the value NULL. Only those attributes of the list widget are updated, where the corresponding call parameters are not equal to NULL.	

FUNCTION: void InOutMethods:: AddMList(char* name, String* List,
int column_number, int distance,
int width, int height)

Parameter:	name:	name of the List widget
	List:	List of the entries
	col_num:	number of columns in the list
	dist:	distance to the previous widget
	width:	width of the widget
	height:	height of the widget
Description:	This method adds a multi list widget to the receiver window. The difference between a list and a multi list is that a user can select more than one entry in the multi list.	

FUNCTION: void InOutMethods::GetMList(char* name, char** value)

Parameter:	name:	name of the List widget
	value:	selected item(s) from the List
Description:	This method gets the selected item(s) from a multi list widget as a list of strings. The selected items are those which are highlighted in the MList window.	

FUNCTION:	void InOutMethods::SetMList(char* name, String* List, char* newlabel, XtCallbackProc callback, caddr_t callData)	
Parameter:	name:	name of the list widget
	List:	pointer to the whole list to add
	newlabel:	set the label of the list window to label
	callback:	associated callback function
	calldata:	data for callback
Description:	This method replaces the attributes of a multi list structure. However only those attributes where the corresponding call parameters are not NULL are replaced.	

FUNCTION:	void InOutMethods::SensitiveDialog(char* name, Boolean state);
	void InOutMethods::SensitiveButton(char* name, Boolean state);
	void InOutMethods::SensitiveMList(char* name, Boolean state);
	void InOutMethods::SensitiveList(char* name, Boolean state);
	void InOutMethods::SensitiveToggle(char* name, Boolean state);	

Parameter:	name:	name of the widget
	state:	TRUE = sensitive, FALSE = insensitive
Description:	This method sets the widget with name 'name' sensitive, if state = TRUE. IF state = FALSE the widget will be set insensitive. The InOut class need some additional methods to control the output window, which are not required by the In class.	

FUNCTION:	void InOut::WriteOut(char* outtext);	
Parameter:	outtext:	text to be appended in the output window
Description:	This method appends the text string 'outtext' to the output window. The new contents of the output window is only displayed, if the output window is in update mode, otherwise it is only stored. You can switch on and off this mode with the two methods WriteOutEnable and WriteOutDisable.	

FUNCTION:	void InOut::WriteOutDisable();
Parameter:	no Parameter
Description:	Disable updates of the output window.

FUNCTION: void InOut::WriteOutEnable();
Parameter: no Parameter
Description: Enable updates of the output window.

2.4 The Message Class

The Message Class can be used to send all kind of messages to the user. These messages can be for example error messages or help texts (figure 2.4). In order to do so, the message class creates a pop-up window that contains the message and an optional interface that enables the user to confirm the message.

3The methods for the class message are divided in three classes besides the constructor and destructor methods. The first set of methods enable the programmer to combine an arbitrary number of Button, Text, Pixmap and Pixlabel widgets to form a message window. The layout algorithm for the different components is as following: Button Widgets are placed from the left to the right within the same “line” and adjusted to the right. The same holds for Pixmap and Pix label widgets, however they are adjusted to the middle of the message window. The first object which is added to the message window starts in the first line. Text widgets form another class they are placed top down in the order they are added. The second class of methods change the contents of an existing message window and the third class of methods show or hide the message window.

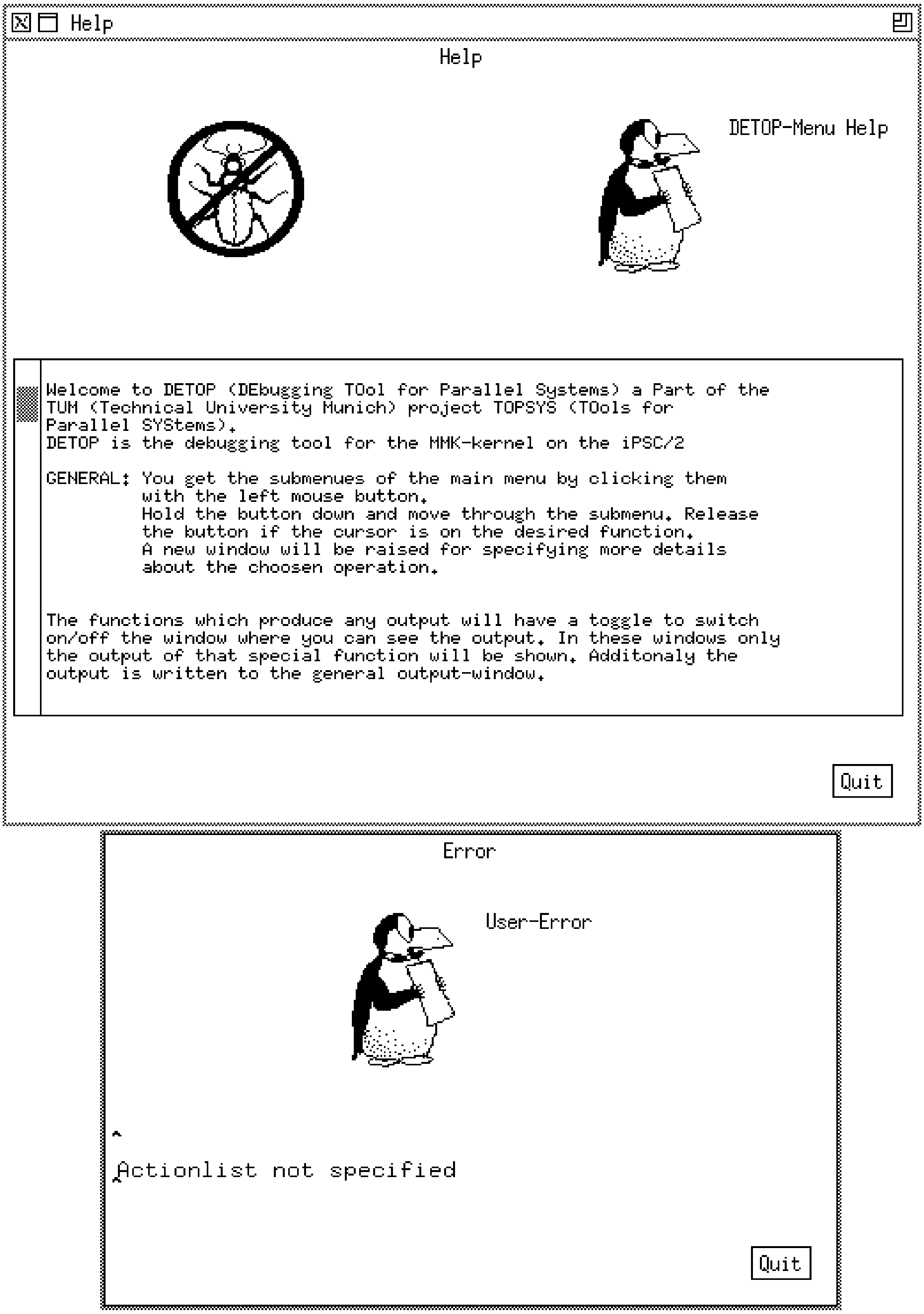


Fig. 2.4: Message class used for building an error window

Methods of the Message Class:

FUNCTION:	Message::Message(char* title, Widget parent, char* font, Pixmap backpix) Message::Message(char* title, Widget parent, char* font, Pixmap backpix, Boolean override)	
Parameter:	title:	title of the input/output window
	parent:	parent widget of the input/output window
	font:	title font
	backpix:	Pixmap for the message window
	override:	creates an override shell, when TRUE.
Description:	The methods Message::Message form the constructor for a message window. They are overloaded methods. The difference is very slight. Using the constructor with the override parameter you can select, if the created Popup shell is an override shell (override = TRUE) or not. The first constructor automatically creates an override shell. Override shells bypass the window manager. One of the initialization methods must be executed before any of the following methods can be used. The constructor first creates a Popup shell widget and a form widget which is used as a container for all other widgets to be added. Then a Label widget with the title of the message window is added.	
FUNCTION:	void Message::~Message();	
Parameter:	-	
Description:	This method is the destructor for a message window. It closes the window and frees all memory.	
FUNCTION:	void Message:: AddButtonPix(Pixmap pixfile, int distance, XtCallbackProc (*call) () caddr_t data)	
Parameter:	pixfile:	Pixmap which is loaded as background for the button
	distance:	Distance to the previous Button (or the border)
	call:	Callback procedure assigned to the button
	data:	Data structure passed to the Callback function
Description:	This method adds a Pix button to a message window and assigns the callback function call to it. The position of the button is determined as following: The first button start in the first empty row. The first empty row depends on the history of the message window. The next buttons are ar-	

ranged in the same row as the first button. The message window is filled from the left to the right.

FUNCTION: void Message:: AddButtonText(char* text, int distance,
char* font,
XtCallbackProc (*call) (),
caddr_t data)

Parameter:

text:	Button text
distance:	Distance to the previous Button (or the border)
font:	font to be used for the button text
call:	Callback procedure assigned to the button
data:	Data structure passed to the Callback function

Description: This method adds a Label button to a message window and assigns a callback function call to it. The method is identical to the previous method with the only difference that the button area is filled with text instead of a pixmap.

FUNCTION: void Message::AddPixPix (Pixmap pixfile, int distance);

Parameter:

pixfile:	Pixmap to be loaded
distance:	Distance to the previous Pixmap

Description: The method AddPixPix adds a Pixmap to the receiver message window. Successive calls fill the message window from the left to the right.

FUNCTION: void Message:: AddPixText(char* text, int distance, char* font);

Parameter:

text:	Text to be displayed
distance:	Distance to the previous pixmap
font:	Font for the text

Description: The method AddPixText adds a Text label to the receiver message window. Successive calls fill the message window from the left to the right.

FUNCTION: void Message:: AddTextFile(3char* textfile, char* font,
int width, int height,int distance);

Parameter:

textfile:	determines the file to be displayed
font:	Font for the text
width:	Width of the text window
height:	Height of the text window
distance:	Vertical distance to the previous text window

Description: The method AddTextString adds a window with scrollbars containing the text string ’textstr’ to the message window. The new window will be located below the previous text window. The vertical distance is determined by 'distance'.

FUNCTION:	void Message:: AddTextString(char* textstr, char* font, int width, int height,int distance)	
Parameter:	textstr:	text to be displayed
	font:	Font for the text
	width:	Width of the text window
	height:	Height of the text window
	distance:	Vertical distance to the previous text window
Description:	The method AddTextString adds a window with scrollbars containing the text string 'textstr' to the message window. The new window will be located below the previous text window. The distance is determined by distance.	

FUNCTION:	void Message:: AlterTitle(char* name);	
Parameter:	name:	new title for the message window
Description:	Replace the title of the receiver window with 'name'.	

FUNCTION:	void Message:: AlterPix(int number, Pixmap pixfile, char* labeltext)	
Parameter:	number:	Number of the Pixmap or Pix label to be replaced
	pixfile:	new Pixfile
	labeltext:	new Pix label
Description:	This method replaces the Pixmap or Pix label which was added as number 'number' with the new Pixmap 'pixfile' or the new Pix label 'labeltext'. Either 'pixfile' or 'labeltext' should be NULL. If both variables are not NULL the 'labeltext' entry is ignored.	

FUNCTION:	void Message:: AlterButton(int number, Pixmap pixfile, char* labeltext)	
Parameter:	number:	Number of the Pix button or Label button to be replaced
	pixfile:	new Pix button
	labeltext:	new Label button
Description:	This method replaces the Pix button or Label button which was added as number 'number' with the new Pix button 'pixfile' or the new Label button 'labeltext'. Either 'pixfile' or 'labeltext' should be NULL. If both variables are not NULL the 'labeltext' entry is ignored. Note that this message does not change the assigned Callback procedure of the button.	

FUNCTION:	void Message:: AlterTextFile(int number, char* textfile)	
Parameter:	number:	Number of the Pix button or Label button to be replaced
	textfile:	new text to be displayed
Description:	This method replaces the text of the text file or text string which was added as number ’number’ with the contents of the file ’textfile’.	
FUNCTION:	void Message::AlterTextString(int number, char* textstr);	
Parameter:	number:	Number of the Pix button or Label button to be replaced
	textstr:	new text to be displayed
Description:	This method replaces the text of the text file or text string which was added as number ’number’ with the contents of the str ’textstr’.	
FUNCTION:	void Message:: Popup(int xpos, int ypos) void Message:: Popupex(int xpos, int ypos)	
Parameter:	xpos:	x position of the message window to pop up
	ypos:	y position of the message window to pop up
Description:	These methods display the receiver message window. The first version displays the window. The second version also grabs the server and forces the user to acknowledge the message.	
FUNCTION:	void Message ::Popdown();	
Parameter:	-	
Description:	This method causes the message window to disappear from the screen.	

2.5 The GenOut Class

The GenOut class creates an output window for text. You can see an example of a GenOut window in figure 2.5. It has a similar functionality as the output part of the InOut class window. The GenOut window can store a large amount of text. It has assigned a scrollbar which enables the user to look at previous outputs. When there is an overflow of the internal text buffer, the first part of it is deleted (about the last 10% of the contents will remain buffered).

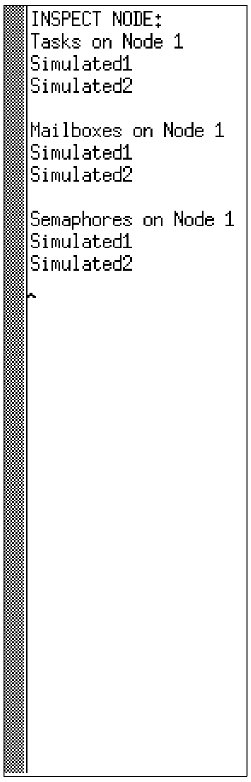


Fig 2.5: GenOut window

Methods of the GenOut Class:

FUNCTION:	GenOut:: GenOut(Widget parent, char* font, int width, int height);	
Parameter:	parent:	Parent widget for the GenOut window
	font:	Font for the genout window
	width:	Width of the genout window
	height:	Height of the genout window
Description:	This method is the constructor of the genout window.	

FUNCTION:	void GenOut:: WriteGenOutput(char* string)	
Parameter:	string:	contains the text to be appended to the GenOut window
Description:	This method appends the string 'string' to the GenOut window. If the output was previously disabled 'string' is only buffered internally but not displayed, otherwise the display is updated immediately. If the storage for 'string' was allocated dynamically you can free it after calling WriteGenOutput.	

FUNCTION: void GenOut::Popup()

Parameter: -

Description: Show the GenOut window on the screen.

FUNCTION: void GenOut::Popdown()

Parameter: -

Description: Pops the GenOut window down.

FUNCTION: void GenOut::WriteGenEnable();

Parameter: -

Description: Enables writing to the GenOut window and updates it, i.e. all output (via WriteGenOutput) while output was disabled is displayed in the genout window.

FUNCTION: void GenOut::WriteGenDisable();

Parameter: -

Description: Disables updates of the GenOut window. All calls of WriteGenOutput will be stored internally and displayed after a WriteGenEnable call. In case of a buffer overflow part of the stored text is deleted (about 10% of the old content of the GenOut window remain in the buffer).

2.6 The Menu and the SpringMenu Classes

Menus are very important to build user interfaces. A menu offers an interface which enables the user to select an entry out of a number of choices. Besides the class menu there is the class SpringMenu, which is used to build pop-up submenus which can be added to menu entries. This section introduces both classes. Figure 2.6 shows the menus which are used by PATOP and VISTOP. PATOP uses labels for its menu entries, VISTOP uses bitmaps. There is also a C interface for the Menu class which is not covered in this document.

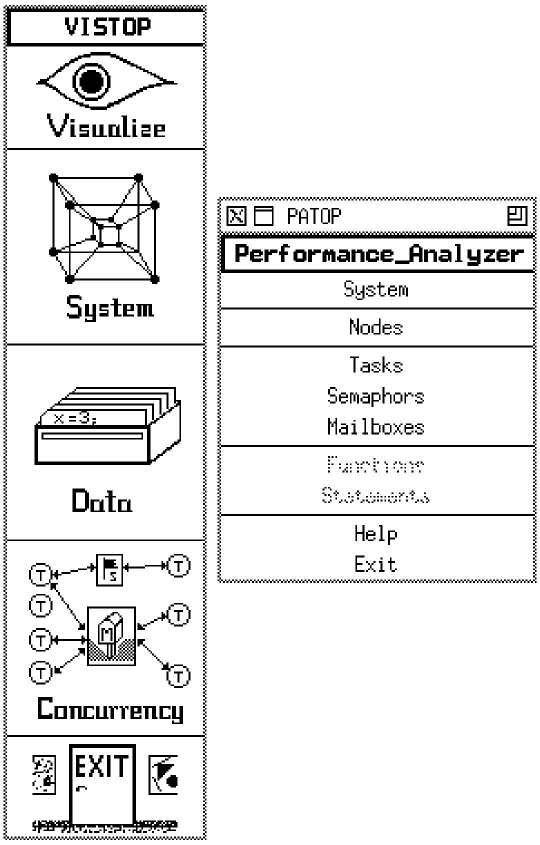


Fig. 2.6: The menus of PATOP and VISTOP

The class Menu has the following public functions:

FUNCTION: Menu:: Menu(char* name, int xpos, int ypos, Boolean iconic,
Pixmap icon, char* icontext, int ixpos, int iypos)

Parameter:
name: Title of the Menu
xpos: Initial x position of the menu (upper left corner)
ypos: Initial y position (upper left corner)
iconic: switch if menu starts up as an icon
icon: Pixmap for the icon
icontext: Name of the icon assigned to the menu
ixpos: Initial x position of the icon (upper left corner)
iypos: Initial y position (upper left corner)

Description: Constructor for the class menu. Creates an empty shell for a menu which can be enhanced by toggles, items, dialogs and lists which are described later.

FUNCTION: Menu:: Menu(char* name, int xpos, int ypos)

Parameter:	name:	Title of the Menus
	xpos:	Initial x position of the menu (upper left corner)
	ypos:	Initial y position (upper left corner)

Description: Second constructor for the menu class. Creates the menu but no icon form. No bitmap will be assigned to the icon form of the menu.

FUNCTION: Menu:: Menu(char* name)

Parameter: name: Title of the Menus.

Description: Third constructor for the menu class. Creates the menu but no icon form. No bitmap will be assigned to the icon form of the menu. The initial position of the menu is 1,1.

FUNCTION: Menu::~Menu()

Parameter: -

Description: Destructor for an instance of class menu.

FUNCTION: void Menu::Popup()

Parameter: -

Description: Shows the menu on the screen. The position of the menu is either the initial position or the last position.

FUNCTION: void Menu::Popup(Boolean Reposition)

Parameter: Reposition: Tells if the menu gets a new location on the screen

Description: Same as the last method if 'Reposition' is FALSE. If however 'Reposition' is TRUE, the new position of the menu is determined by the mouse position.

FUNCTION: void Menu::Popdown()

Parameter: -

Description: Remove menu from screen. Nothing of the menu will be destroyed.

FUNCTION: void Menu::Prepare()

Parameter: -

Description: This function will be called implicitly the first time a pop-up method is executed. This method calculates the size of the menu and centers all entries. Afterwards no further entries may be added to the menu.

FUNCTION: void Menu::AddToggle(char* name, XtCallbackProc (*SetCb) (Widget, caddr_t, caddr_t), caddr_t set_data, XtCallbackProc (*UnsetCb)(Widget,caddr_t,caddr_t) caddr_t unset_data)

Parameter:

name:	Name and Label for the toggle button
SetCb:	Callback to be executed when button is set
set_data:	Data for SetCb
UnsetCb:	Callback to be executed when button is unset
unset_data:	Data for UnsetCb

Description: This method adds a Toggle button to the receiver menu. This button gets two callback functions assigned. The first is called when the button is set, the second when it is unset. The toggle is placed below the previous element of the menu.

FUNCTION: void Menu::AddToggle(char*name, Pixmap pixmap, XtCallbackProc (*SetCb) (Widget, caddr_t, caddr_t), caddr_t set_data, XtCallbackProc (*UnsetCb)(Widget,caddr_t,caddr_t) caddr_t unset_data)

Parameter:

name:	Name of the toggle button
pixmap:	Pixmap for the background of the button
SetCb:	Callback to be executed when button is set
set_data:	Data for SetCb
UnsetCb:	Callback to be executed when button is unset
unset_data:	Data for UnsetCb

Description: This method is similar to the previous method. However instead of a text label, the button get a pixmap.

FUNCTION: void Menu::AddItem(char* name, XtCallbackProc (*Cb)(Widget,caddr_t,caddr_t) caddr_t data)

Parameter:

name:	Label and name of the button
Cb:	Callback function which is executed when a button is pressed
data:	Data for Cb

Description: This method adds a Choice Button to the receiver menu.

FUNCTION: void Menu::AddItem(char* name,Pixmap pixmap,
XtCallbackProc (*Cb) (Widget, caddr_t, caddr_t)
caddr_t data)

Parameter:
- name: Name of the button
- pixmap: Pixmap which is loaded in the background of the button
- Cb: Callback function which is executed when a button is pressed
- data: Data for Cb

Description: This method adds a Choice Button to the receiver menu. The button gets a pixmap loaded in the background.

FUNCTION: SpringMenu *Menu::AddMenu(char* name, char* name_submenu)

Parameter:
- name: Name of the menu button name
- submenu: Name of the submenu
- Return: Pointer to an instance of a submenu

Description: This method adds a button to a Menu which gets assigned a spring loaded submenu. When the button is pressed the submenu is popped up at the current cursor position. The submenu itself can be filled using the appropriate Spring menu methods.

FUNCTION: SpringMenu *Menu::AddMenu(char* name, Pixmap pixmap,
char* name_submenu)

Parameter:
- name: Label and Name of the button
- pixmap: Pixmap to be loaded in the button
- name: submenu Name of the spring loaded submenu.
- Return: Pointer to an instance of a submenu

Description: This method is similar to the previous. The only difference is the fact, that a pixmap is loaded as background of the button instead of a text label.

FUNCTION: void Menu::AddList(har* name, int min_height, int min_width,
Boolean radio_button)

Parameter:
- name: Name and Label of the list
- min_height: Minimal height of the list in lines. This is the real height of the list, if it doesn’t get values assigned before the prepare is executed.
- min_width: Minimal width of the list. This is the real width of the list, if it doesnt get values assigned before the prepare is executed.

	radio_button:	This parameter decides, if the list elements have the properties of a radio button, i.e. if exactly one item can be selected.
Description:	This method adds a list with the title 'name' to the receiver menu.	

FUNCTION:	void Menu::AddList(char* name, int min_height, int min_width, Boolean radio_button,void (*callback)(char**))	
Parameter:	name:	Name and Label of the list
	min_height:	Minimal height of the list in lines. This is the real height of the list, if it doesn't get values assigned before the prepare is executed.
	min_width:	Minimal width of the list. This is the real width of the list, if it doesnt get values assigned before the prepare is executed.
	radio_button:	This parameter decides, if the list elements have the properties of a radio button, i.e. if exactly one item can be selected.
	callback:	Callback, which is executed when one of the list elements is selected. It gets as call parameter the list of selected objects.
Description:	This method adds a list with the title 'name' to the receiver menu and assigns a callback function to the list which is executed, whenever an element of the list is selected.	

FUNCTION:	void Menu::AlterList(char* name, char **liste)	
Parameter:	name	Name of the list
	liste	List elements which are to be filled into the list. The last element of the list has to be NULL
Description:	This method replaces the list of the list structure with name 'name'with the elements of 'liste' which is an array of strings.	

FUNCTION:	char** Menu::ReadList(char* name)	
Parameter:	name	Name of the list
Return:	List of char strings which is terminated by a NULL-Pointer. It contains all the selected elements of the list 'name'	
Description:	Method that returns all selected elements of a list.	

FUNCTION:	void Menu::AddDialog(char *name, char* initvalue)	
Parameter:	name:	Name of the Dialog window
	initvalue:	Initial value of the window
Description:	This method adds a Dialog window which can be edited. The size of the text window is determined by the length of the initial text.	
FUNCTION:	char* Menu::ReadDialog(char *name)	
Parameter:	name:	Name of the Dialog window
	Return:	String containing the current value of the Dialog window.
Description:	This is the counterpart method to AddDialog. It reads the contents of the Dialog window 'name'.	
FUNCTION:	void Menu::AddSeperator()	
Parameter:	-	
Description:	This method adds a line below the last element that was added to the menu. Without AddSeparator, all menu entries are packed together without break in between.	
FUNCTION:	void Menu::SetToggle(char* name)	
Parameter:	name:	Name the toggle button
Description:	This method sets a toggle button and calls the assigned callback function.	
FUNCTION:	void Menu::UnsetToggle(char* name)	
Parameter:	name:	Name the toggle button
Description:	This method resets a toggle button and calls the assigned callback function.	
FUNCTION:	void Menu::Insensitive()	
Parameter:	-	
Description:	This method sets the whole menu in the state insensitive.	
FUNCTION:	void Menu::Insensitive(char* name)	
Parameter:	name:	Name of the button
Description:	This method sets the button with the name 'name' in the state insensitive.	

FUNCTION: void Menu::Sensitive()

Parameter: -

Description: This method sets the whole menu in the state sensitive.

FUNCTION: void Menu::Sensitive(char* name)

Parameter: name: Name of the button

Description: This method sets the button with the name 'name' in the state sensitive.

The class SpringMenu has the following public functions:

FUNCTION: SpringMenu::~SpringMenu()

Parameter: -

Description: This method is the destructor for a pop-up menu. You will probably notice that there is no constructor for class SpringMenu. The constructor is called implicitly by the methods AddMenu of the class Menu. This is the only possibility to create an instance of class SpringMenu. The destructor removes the SpringMenu from the screen and frees all the dynamic allocated data.

FUNCTION:
```
void SpringMenu::AddItem( char* name,
                          XtCallbackProc (*Cb)(Widget,caddr_t,caddr_t)
                          caddr_t data)
```

Parameter:
- name: Label and name of the button
- Cb: Callback function which is executed when a button is pressed
- data: Data for Cb

Description: This method adds a Choice Button to the receiver SpingMenu.

FUNCTION:
```
void SpringMenu::AddItem( char* name,Pixmap pixmap,
                          XtCallbackProc (*Cb) (Widget, caddr_t,caddr_t)
                          caddr_t data)
```

Parameter:
- name: Name of the button
- pixmap: Pixmap which is loaded in the background of the button
- Cb: Callback function which is executed when a button is pressed
- data: Data for Cb

Description: This method adds a Choice Button to the receiver SpringMenu. The button gets a pixmap loaded in the background.

FUNCTION: void SpringMenu::AddSeperator()

Parameter: -

Description: This method adds a line below the last element that was added to the SpringMenu. Without AddSeparator, all menu entries are packed together without break in between.

FUNCTION: void SpringMenu::Insensitive()

Parameter: -

Description: This method sets the whole SpringMenu in the state insensitive.

FUNCTION: void SpringMenu::Insensitive(char* name)

Parameter: name: Name of the button

Description: This method sets the button with the name 'name'in the state insensitive.

FUNCTION: void SpringMenu::Sensitive()

Parameter: -

Description: This method sets the whole SpringMenu in the state sensitive.

FUNCTION: void SpringMenu::Sensitive(char* name)

Parameter: name: Name of the button

Description: This method sets the button with the name 'name' of the SpringMenu in the state sensitive.

FUNCTION: void SpringMenu::Prepare()

Parameter: -

Description: This function will be called implicitly the first time the SpringMenu is popped up. This method calculates the size of the menu and centers all entries. Afterwards no further entries may be added to the menu.

2.7 C++ Interface for the Athena Widgets

The X Window System follows the object oriented programming paradigm. The GUI of the tools is mostly implemented in C++, which supports this paradigm in an adequate way.

In X11 R4 the Xlib and the XtIntrinsic routines have function prototypes defined, however not the widgets from the Athena widget set. So we replaced the public header files of all the widgets described in this document with header files containing function prototypes like those in Xlib. They can be found in the directory /usr/proj/ipsc2/TOPSYS/HOST/GRAPHIK/INCLUDE/CC. Their names are constructed from the names of the Athena widget header files by adding 'CC' before the .h (i.e. the C++ include file for a Label widget is called LabelCC.h instead of Label.h).

Part 3:

The Tools DETOP, PATOP, VISTOP

Peter Braun

This part of the design documents describes the top layer of each of the TOPSYS tools DETOP, PATOP and VISTOP. The tools are based both on the graphical objects described in part 2 and on the HLTL interface described in part 4. The graphical objects perform the basic components for the GUI (graphical user interface), the HLTL allows the communication of the tools with an MMK application running on a parallel computer. This part assumes that you are familiar with parts 2 and 4 of the design documents. The top layer of the tools also depends on the X Window System, i.e a certain knowledge of the underlying concepts might be helpful.

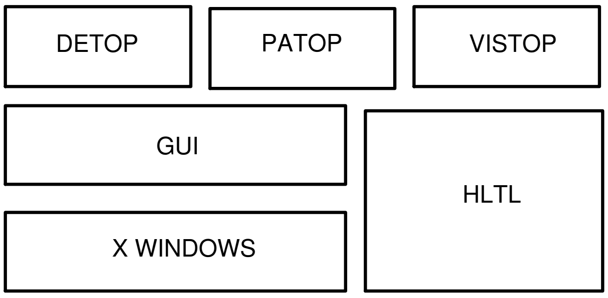


Fig. 1.1: Hierarchical architecture of TOPSYS tools

1 DETOP

In the first chapter we describe the implementation of the GUI of the parallel debugger DETOP. It assumes that you are familiar with DETOP's functionality which is described in the "DETOP User's Guide".

The DETOP's GUI is based on the graphical classes like MENU, MESSAGE and GENOUT. however it's functionality is based on the routines implemented in the UTOL layer (i.e. the HLTL interface). The HLTL functions enable DETOP to start and run applications, to specify breakpoints and traces, to inspect variables and to do other useful things. In general, when a user specifies some action in one of DETOP's graphical objects, DETOP calls the appropriate HLTL function(s) which interact(s) with the application that is to be debugged. DETOP is more or less only a shell which offers the user a comfortable and comprehensive interface to the debugging capabilities offered by the HLTL functions.

1.1 Communication

All communication between DETOP and an application running on the parallel computer is performed by the HLTL functions, i.e. DETOP's graphical user interface does not even know that the application is running on a different machine. Most of the communication between DETOP and an application is synchronous. Whenever DETOP wants to interact with an application, DETOP calls the appropriate HLTL function (e.g. inspect_s5()) and is suspended until the result of the query is present. The only asynchronous communication takes place when working with breakpoints. When an application reaches a specified breakpoint, it notifies DETOP asynchronously via a handler function. DETOP assigns the handler function when calling the break definition routine dbreak_s5().

1.2 Breakpoint Notification

The file handler.C contains the functions which implement the break handler. This module consists of three functions:

- the break handler function (b_handler()), which is called asynchronously, when a breakpoint is reached. The handler is assigned in the file BREAK/Break.C by the function dbreak_s5(). As you may not release X Window requests asynchronously, the handler only writes to a pipe. This pipe is added as another event source for DETOP.

- the initialization function (init_handler()); it tells DETOP to call the procedure br_ev_handler(), when an input to the pipe occurred (i.e. a breakpoint was reached). This function is called in the main loop (main.c).
- the notifier function br_ev_handler() it creates the break message window (see figure 2.4 of part 2)

1.3 Module Structure of DETOP

The code for DETOP is located in /usr/proj/ipsc2/TOPSYS/HOST/TOOLS/DETOP and the directory tree below this path. However this location is not mandatory. All subdirectories stated in this chapter are relative to the DETOP root directory.

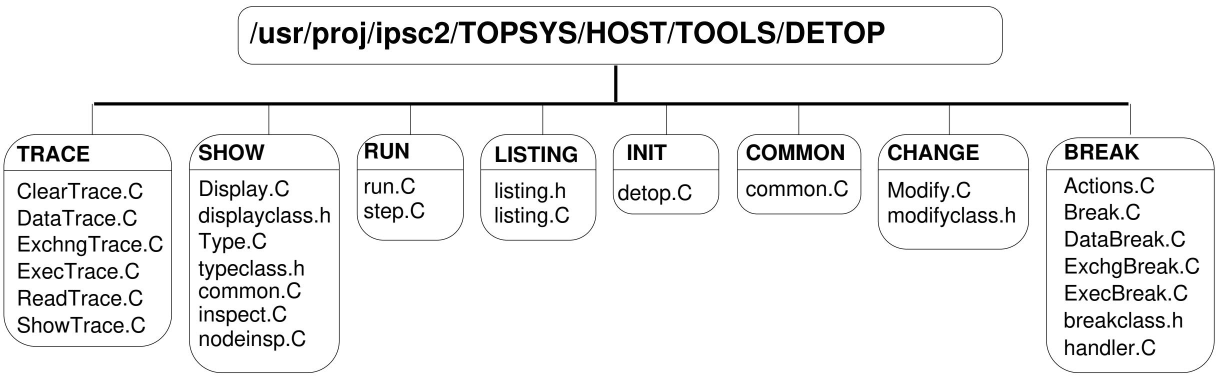


Fig. 1.2: DETOP modules

Most of the modules of DETOP are used to implement the various graphical objects, which form the shell for all of DETOP's debug capability. In order to invoke graphical objects from a sub-menu, DETOP uses two different ways:

1. The callback function assigned to a menu button creates a new window using the classe described in part 2 (refer to the code of SHOW/inspect.C for an example how this is performed).
2. The callback function assigned to the menu button calls the constructor method of the graphical object. For all objects being created this way, there is a class description file which contains the class interface. (refer to SHOW/Display.C for an example how this is performed. The class definition file for the display window is in SHOW/displayclass.h).

Both approaches offer the same functionality to the user and the design decision is simply a matter of taste of the authors. There is a naming convention to distinct both approaches. The files using the first method start with a lower case letter (e.g. inspect.C). The files implementing graphical objects by the second method start with an upper case letter (e.g. Display.C).

Note:

When you have a look at DETOP's source code, you might wonder to see a lot of function interfaces like example_callback(Widget w, caddr_t value1 ,caddr_t value2), where some or all of the parameters are unused. These are callback functions. The interface of a callback function is determined by X Windows. See your X Window documentation how X Window passes information and allows you to pass information to a callback function.

1.3.1 DETOP Main Loop

The main module (main.c) is located in the DETOP root directory. This module is the main loop for DETOP. It initializes the X windows toolkit, opens the display connection, initializes the UTOL, initializes the break handler, and calls init_detop().

1.3.2 Initialization

Initialization of the graphical interface is performed by the function init_detop(). This function is contained in the file INIT/detop.C. This is the first of DETOP's GUI functions to be

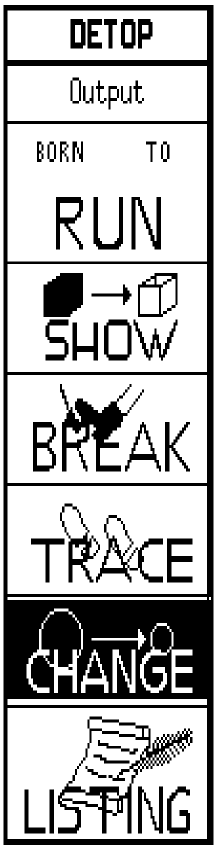


Fig. 1.3: DETOP Main menu

executed. It initializes the help and error facilities and creates DETOP's initial menu (figure 1.3). In addition it assigns popup sub-menus to the RUN, SHOW, BREAK, TRACE and CHANGE button of the main menu and assigns callback functions to all the buttons of the popup sub-menus and the remaining buttons of the main menu. You select an item of one of the sub-menus by moving the mouse pointer over the item and pressing the left mouse button.

1.3.3 Graphical Objects

DETOP's most interesting graphical objects from the point of view of the user, are the windows that may be created from the different sub-menus. Each entry of each sub-menu has assigned a callback function which creates a window, where the user can specify some parameters. These parameters specify actions and objects of the application. When you press the 'OK' button in one of the windows, the appropriate HLTL function is called, which specifies a breakpoint, displays a variable, lists a trace...

All windows of DETOP are created only once then they are only removed from the screen but not destroyed (i.e. managed and unmanaged) . This is much faster than creating and destroying objects every time they appear on or disappear from the screen. For this purpose, each object has a private boolean variable named 'prepared' which is FALSE, until the object is created.

DETOP's windows are created using the graphical objects described in part 2. The implementation for all the functions assigned to the buttons is a bit lengthy but nevertheless straightforward. So the rest of the chapter gives only a short description of the contents of each module in form of a table and presents screen dumps of some of DETOP's windows. Each of the following tables names the "constructor" callback function of the graphical object and the module name where the function is implemented. Even in the cases where an object is created by the second method, the callback function, which is called form the sub-menu is named in the table. This function calls the constructor method for the object. This should give you enough background to look up details by yourselves in the source code.

Sub-Menu RUN

This part of DETOP starts an application, part of an application or steps through the application in single step mode. The code for the callbacks of the 'RUN' popup menu entries 'ALL', 'Tasks' and 'Steps' are contained in the sub-directory RUN.

The windows which are created by selecting an entry from the 'RUN' sub-menu are shown in figure 1.4.

Entry	Action	Callback function	File
ALL	start all tasks	dgo_all	Run.C
Tasks	start selected tasks	d_go	Run.C
Step	single step	d_step	Step.C

Table 1.1 table of the graphical objects of RUN

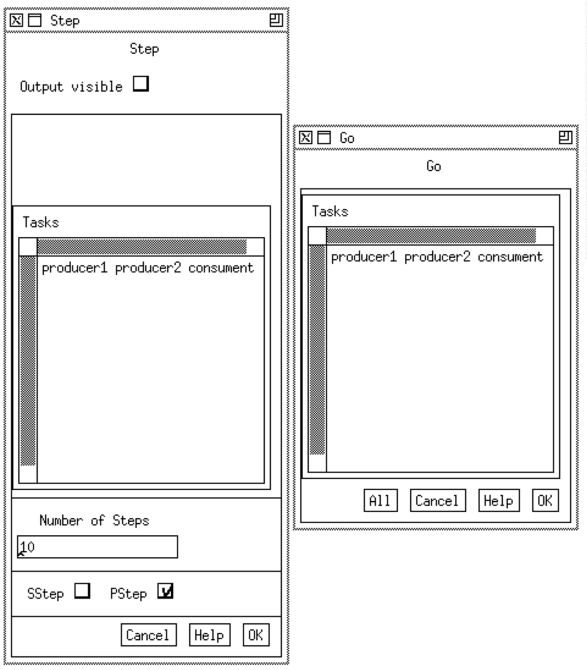


Fig. 1.4: RUN windows Go and Step

Sub-Menu SHOW

The SHOW sub-menu contains DETOP's functions to inspect and display objects of the application. It has the entries 'INSPECT' and 'INSPECT NODES', 'TYPE' and 'DISPLAY'. 'INSPECT' inspects the state of the MMK objects of the application, 'INSPECT NODES' shows the MMK objects located on a specific node. 'TYPE' can be used to determine the type of a variable and 'DISPLAY' to determine the value of it.

Figure 1.5 shows the windows which are created by selecting one of the entries of the 'show' sub-menu

Entry	Action	Callback	File
INSPECT	inspect state of MMK objects	d_insp	inspect.c
INSPECT NODES	show MMK objects of nodes	d_node	nodeinsp.c
TYPE	show type of variable	d_type	Type.c
DISPLAY	display value of variable	d_display	Display.c

Table 1.2 table of the graphical objects of SHOW

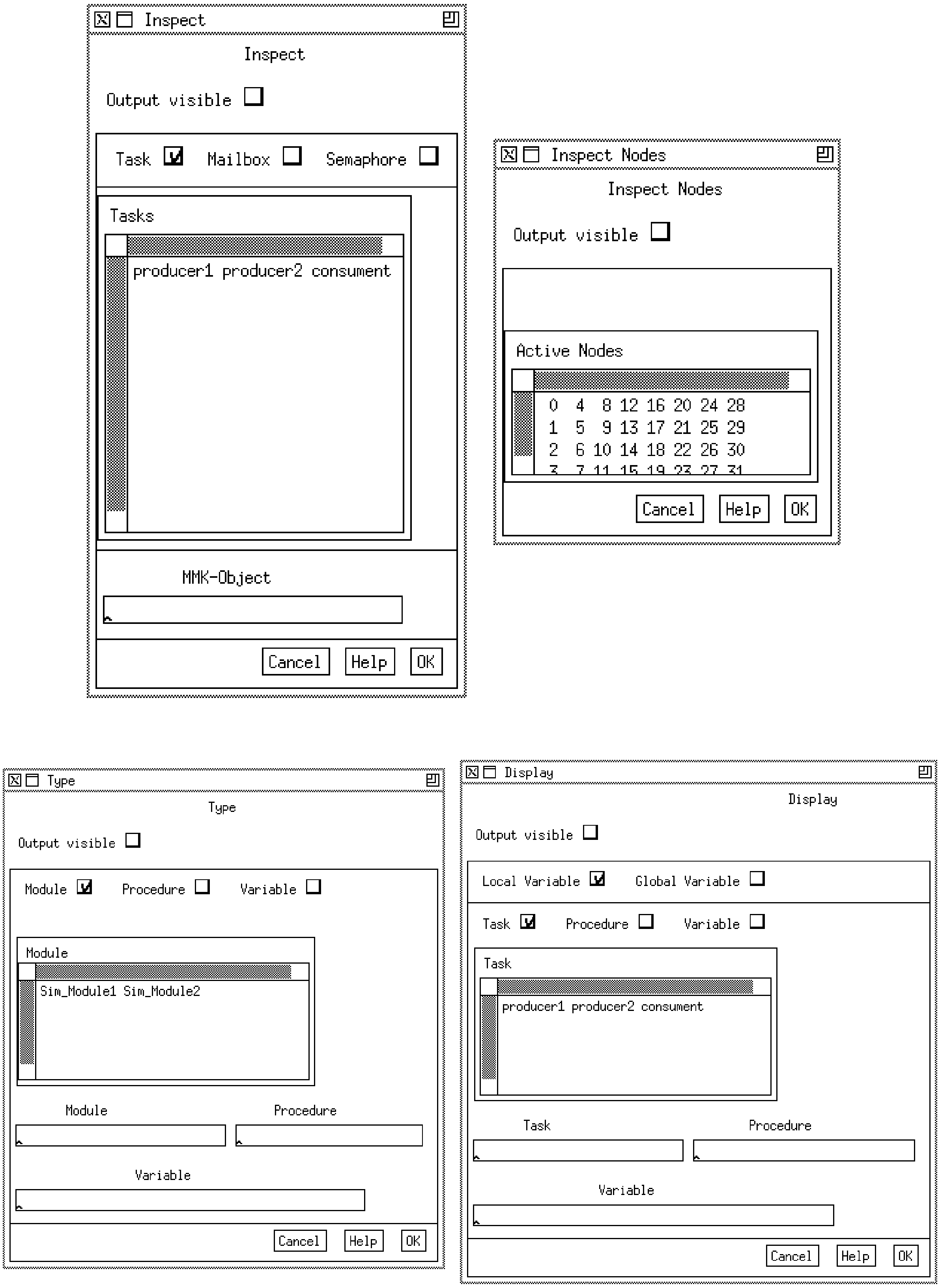


Fig. 1.5: SHOW: inspect, inspect nodes, type, display

Sub-Menu BREAK

The code for all objects of this sub-menu is located in subdirectory BREAK. These objects let you specify the three different type of breakpoints supported by DETOP. These are data, execution and exchange breaks. The windows being created by the entries of the BREAK menu are shown in figure 1.6.

Entry	Action	Callback	File
Define Breakpoint	specify break predicate	d_break	Break.c
Clear Breakpoint	clear break specification	d_clear_break	Break.c
Master Break	stop all processes	d_master_break	Break.c

Table 1.3 table of the graphical objects of BREAK

The special thing about DETOP's 'Break' window, which is created by pressing the 'Define Breakpoint' button, is the fact, that it also invokes new graphical objects to specify data, execution and exchange breakpoints and to specify the action which has to be performed, when the break predicate is satisfied in the application. The way you can specify breakpoints reflects how that DETOP builds more complex break predicates out of a sequence of simple break predicates. The additional graphical objects invoked from the define breakpoint window are listed in table 1.4. Note that the three Break windows are created via a constructor function, the action window is created by a callback function.(see figure 1.6)

Break Window	Action	Constructor	File
Data Break	specify data break	DBreak_class	DataBreak.C
Execution Break	specify execution break	ExBreak_class	ExecutionBreak.C
Exchange Break	specify exchange break	ExchBreak_class	ExchangeBreak.C
Action	specify actions	db_Action_Buttons	Action.c

Table 1.4 graphical objects invoked by the define breakpoint window

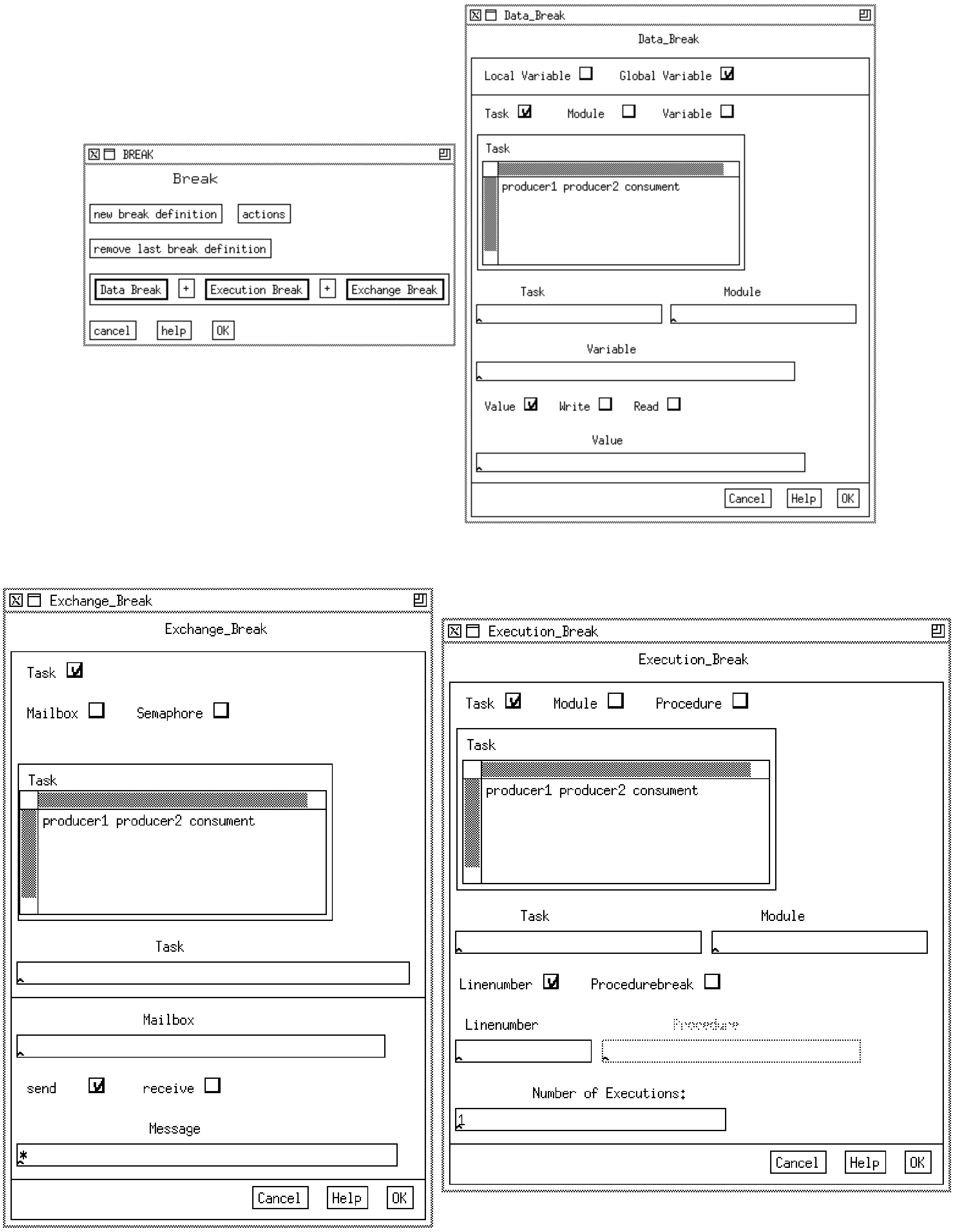


Fig. 1.6: BREAK windows

Sub-Menu TRACE

This sub-menu contains the entries to create and manage traces, i.e. to specify Data, Execution and Exchange Trace, to read the traces and to remove them. The functions implementing these routines are contained in subdirectory TRACE (figure 1.7).

Entry	Action	Callback	File
Execution Trace	specify break predicate	d_data_trace	DataTrace.C
Execution Trace	stop all processes	d_exec_trace	ExecTrace.C
Exchange Trace	clear break specification	d_exch_trace	ExchTrace.C
List Traces	list all specified traces	d_list_trace	ShowTrace.C
Clear Trace	clear specified traces	d_clear_trace	ClearTrace.C
Trace Buffer	display trace contents	d_read_trace	ReadTrace.C

Table 1.5 table of the graphical objects of TRACE

The windows being created by the entries of the TRACE menu are shown in figure 1.7.

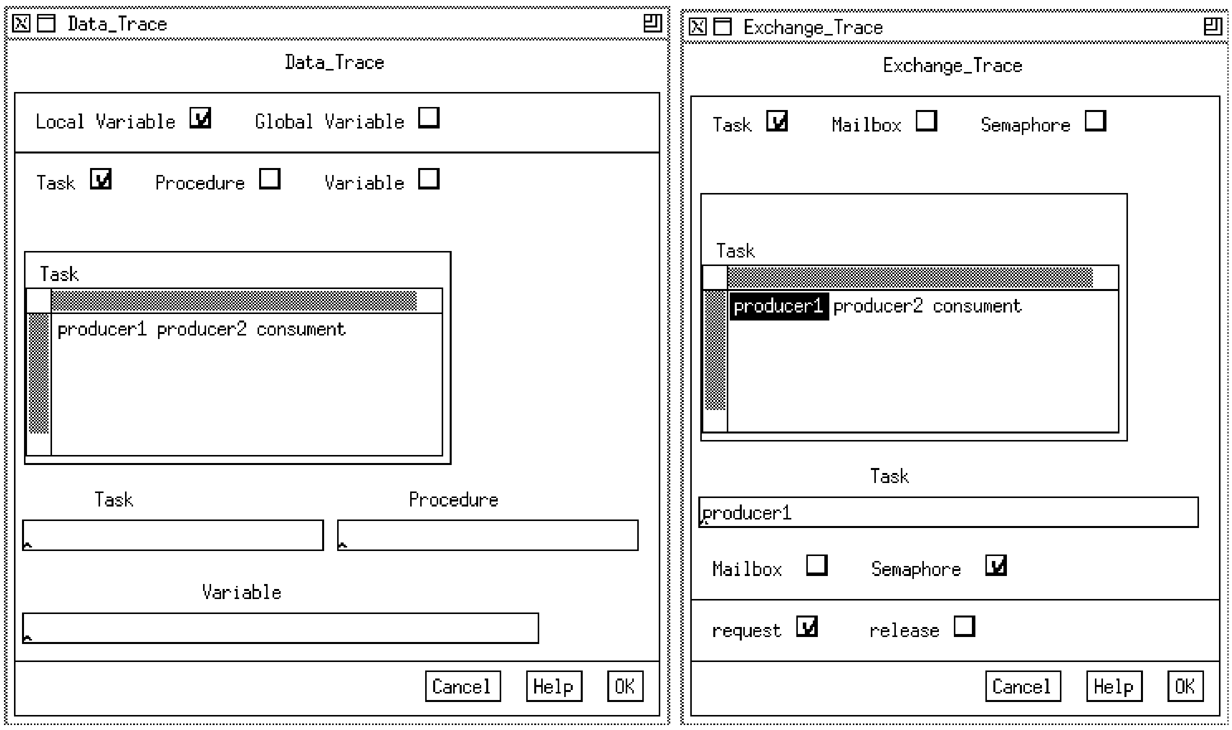


Fig. 1.7: TRACE window

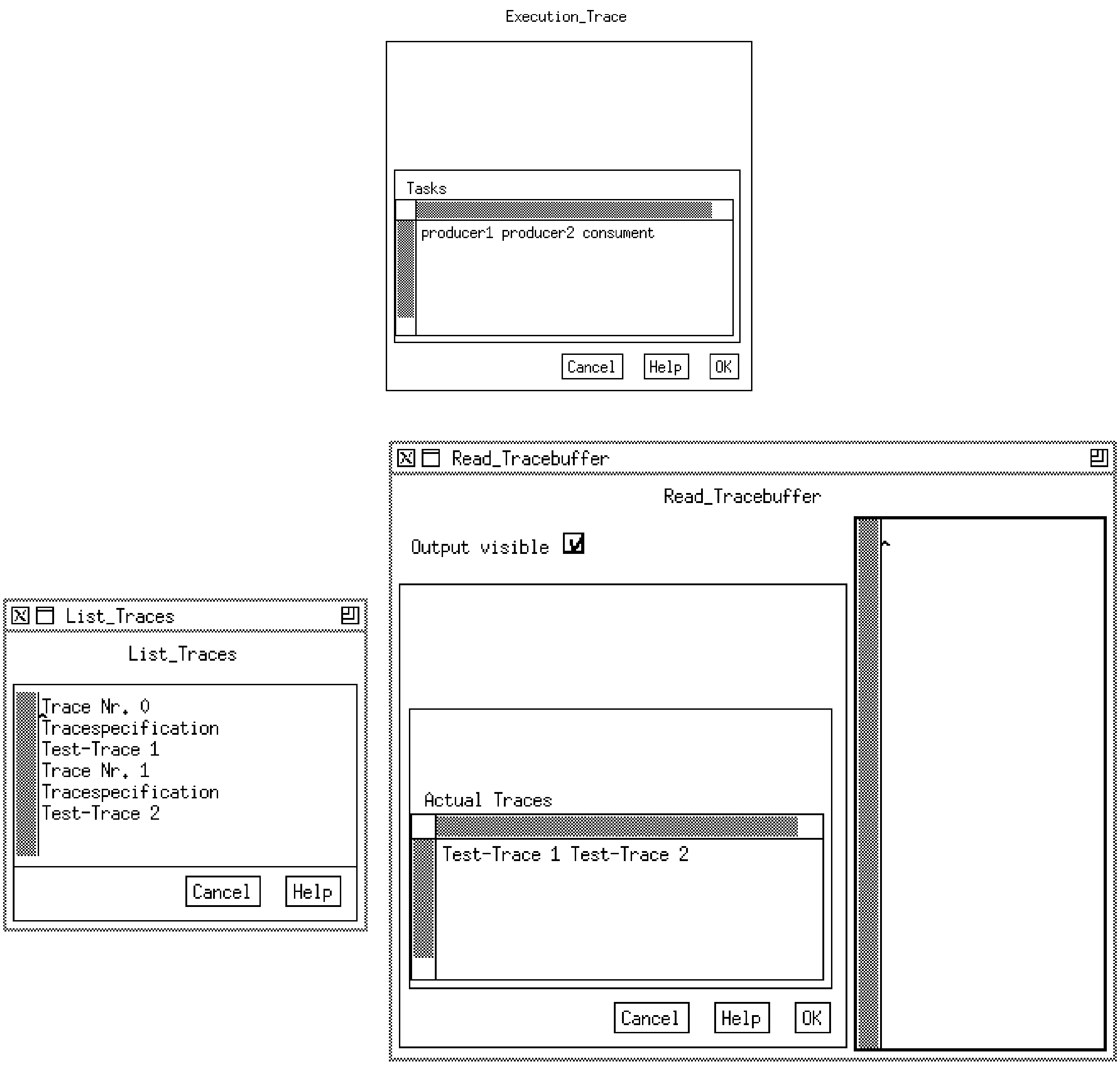


Fig. 1.7 (cont.): TRACE windows

Sub-Menu CHANGE

This sub-menu only has the entry Modify. It allows you to change the value of variables. The modify window is shown in figure 1.8.

Entry	Action	Callback	File
Modify variable	change value of a variable	d_modify	Modify.C

Table 1.7 table of the graphical objects of CHANGE

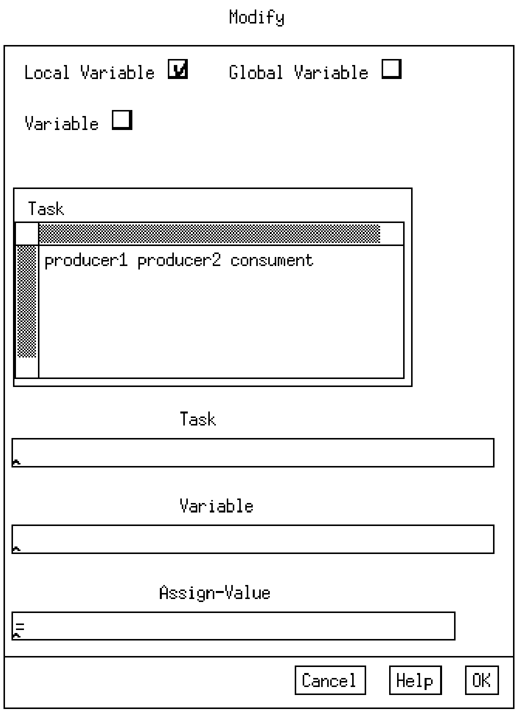


Fig. 1.8: CHANGE window 'modify'

Sub-Menu LISTING

When you press the LISTING button of the DETOP main menu, no sub-menu is created but a dialog box (see figure 1.9), where you can enter the name of a source file. This file is displayed in a text window.

Fig. 1.9: LISTING dialog box

Sub-Menu HELP

DETOP's help facilities consist of the help texts contained in directory HELP and the functions init_detop_help() and reset_detop_help() of the module COMMON/Common.C

A help window is shown, when you press a HELP button from DETOP's main menu or from one of the other windows. The help window itself is created in INIT/detop.C.

2 PATOP

This chapter describes the implementation of the graphical user interface (GUI) of the performance analyzer tool PATOP. PATOP's functionality is based on its HLTL interface described in part 4. For its GUI PATOP uses the graphical objects presented in part 2 of the design documents.

2.1 Performance Data

The UTOL contains four functions which are used by PATOP to collect performance data of the application which is monitored. These are the procedures pa_define (which specifies a performance measurement), pa_start (to start a measurement), pa_read (to read new measurement values) and pa_remove (to stop a measurement). These functions are described in part 4 of the design documents. Internally these functions use different methods to collect data (traces, counters and integrators), at their user interface however, they offer the same functionality. Traces, counters and integrators are described in part 5 of the design documents.

The function pa_read() returns an object of type Pa_value. This is a list of measurement values, where each list element contains

- a measurement value
- the start time of the measurement
- the end time of the measurement

In the performance windows (bargraph or diagram) each value appears scaled by the time interval that comprises the measurement.

PATOP evaluates the Pa_value lists in the procedure update() (OUTPUT/measure.C). This function is called by a timer. The time-out interval of the timer is determined by the update value of the measurement start menu (see figure 2.4).

2.2 Measurements

To understand how PATOP is implemented you have to look how PATOP defines and stores the various measurements a user specifies. This is performed by the classes described in INCLUDE/patop_types.h. It contains the interface of the classes Measure and Startup. See the interface definition for these classes at the end of this chapter for details.

The data displayed in one of PATOP's performance windows can consist of more than one measurement, even if these different measurements may be displayed together in one bar or diagram.

These measurements that are displayed together are combined in structure called Op_list. An Op_list is created by the methods of class Startup. The method Startup::start() finally creates an instance of class Measure containing all measurements contained in the Op_list.

Specification of a measurement

PATOP specifies a measurement or a list of measurements in four steps:

1. All desired measurements are entered in an instance of Class Startup, which itself is based on a data structure called Op_list. This data structure contains for each measurement the important attributes, e.g. which objects (nodes, mailboxes ...), which kind of measurement (idle time, time in a waiting queue ...), which graphical representation (bargraph or diagram).
2. The information specified in a Startup instance is used to define the measurements in the application. This is done by creating an instance of class Measure (Startup::start()).
3. A setup_menu for the measurement is created, where the granularity and update intervals for the measurements can be entered.
4. When selecting the 'start' button of the setup menu, the measurement(s) are defined (pa_define()) and started (pa_start()) by the method measure::start(). This method creates the window for the measurement(s) and starts a timer, which calls the update function for the measurement window.

Displaying the measurements

Patop uses the classes HISTO and BARGRAPH to display the performance measurement values. These classes create curve and bargraph diagrams (see part 2 for details and examples). PATOP's GUI simply uses the methods of these classes to create the appropriate diagrams.

Updating the measurement diagrams

After a measurement is started, the performance measurement window is updated automatically by timer. The timer is implemented with the X Toolkit function XtAddTimeOut(), i.e. each window is updated independently within the function update(). This function first reads new performance measurement values (pa_read()). If more than one measurement has to be combined for one window, update() averages the values for a curve diagram or separates the values for a multi bargraph diagram.

The code for specifying measurements is contained in the files of subdirectory INIT. The code for reading measurement values and updating the performance windows is contained in OUTPUT/measure.C.

2.3 Module Structure of PATOP

The code for PATOP is located in the tree beyond the subdirectory /usr/proj/ipsc2/TOPSYS/HOST/TOOLS/PATOP. Most of the top level of PATOP's (GUI) is implemented in C++. A small part is implemented in C.

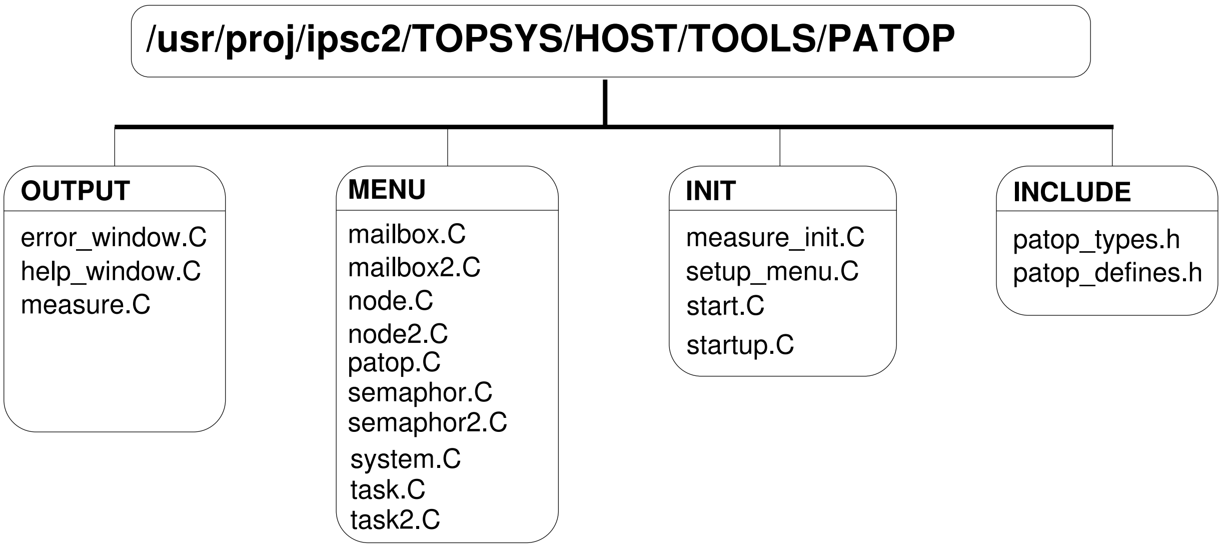


Fig. 2.1: PATOP module structure

PATOP's source code is partitioned in three sub-directories beyond the PATOP root directory. The subdirectory INIT contains the programs to specify a measurement and to start it. This is the base for all of PATOP's functionality. The second directory menu builds up the GUI. The different Menus use the functionality provided by the functions of INIT to specify and finally start the various measurements. The subdirectory OUTPUT contains the file Measure.C, which fills the measurement windows with the values received by the function pa_read().

This part is not intended as a reference guide for PATOP. It presents only the most important objects of PATOP and should give you the background to explore the source code by yourselves.

2.3.1 PATOP Main Loop

This is the main routine for PATOP. It opens the display connection to the server, initializes the UTOL and PATOP and starts the application. Finally it contains the application main loop for dispatching the X events.

2.3.2 Initialization

The initialization of PATOP's graphical interface is performed by the function init_patop(). This function is contained in the file MENU/patop.C. It is the first of PATOP's GUI functions to be executed. It initializes the help and error facilities and creates PATOP's initial menu (figure 2.2). In addition it assigns popup sub-menus to the System, Nodes, Tasks, Semaphors and Mailboxes button of the main menu and assigns callback functions to the buttons of the popup sub-menus. The help button gets a help message window assigned. The Exit button stops the PATOP and the application which is monitored. You select an item of one of the sub-menus by moving the mouse pointer over the item and pressing the left mouse button.

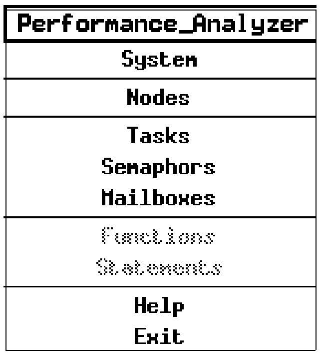


Fig. 2.2: PATOP main menu

2.3.3 Menus

PATOP uses a lot of different menus to enable the user to specify his/her desired measurements.

Most of the source code for menus is located in the subdirectory MENU. All the menus are based on the class Menu (see part 2 of the TOPSYS design documents). Each of the files contains the code to create the menu and the code for the callback functions assigned to the buttons of the menu entries. These callback functions use the methods of class Startup to define measurements. In order to start a measurement the setup_menu is created. The implementation of all the menus is straightforward and is not described here.

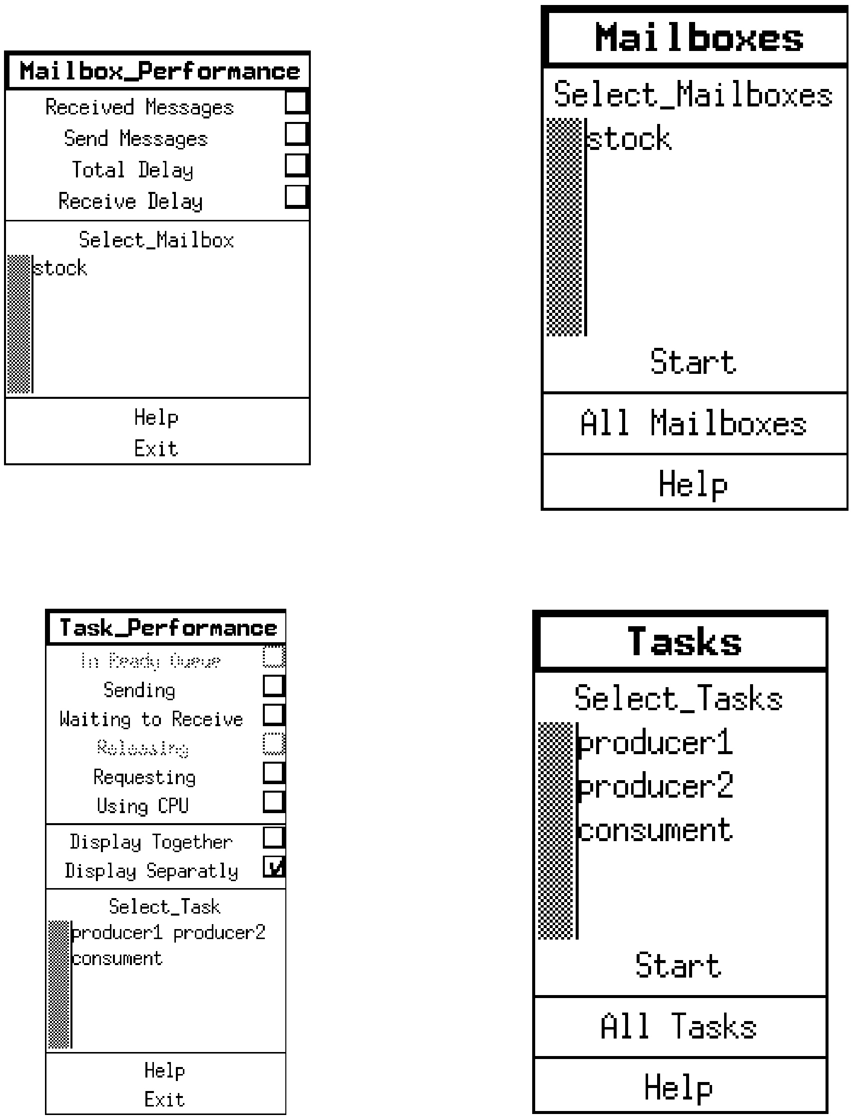


Fig. 2.3: A selection of PATOP's menus

The setup menu (figure 2.4) is implemented in the file INIT/Setup.C. It belongs logically to the source code to specify a measurement. A setup menu is created before a measurement can actually be started. The title of the setup menu shows the kind of measurement to be started.

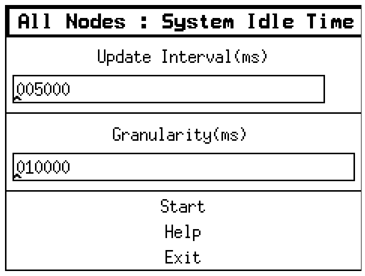


Fig. 2.4: PATOP setup menu

2.3.4 Help Windows

The file help_window.C provides the help facility for PATOP. Each menu has a help button assigned. When the help button is selected, a help_window is created displaying a help message which depends on the context. The help window itself is based on the class Message (see part 2). A sample help window can be seen in figure 2.5. Before you can execute the procedure help_window(), you first have to call init_help() which is called by init_patop().

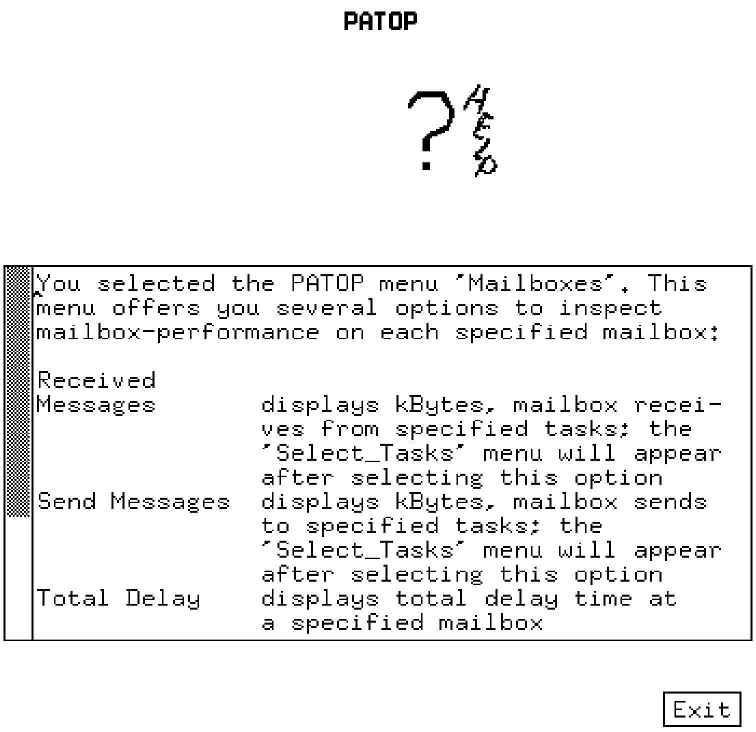


Fig. 2.5: PATOP help window

The help texts being displayed in the help windows are contained in the sub-directory HELP.

2.3.5 Error Windows

Error windows are created via the files of the TOPCOM subdirectory (see chapter 4 of the design documents).

2.3.6 Bitmaps

The bitmaps which are displayed in the error and help window are contained in the directory BITMAPS.

2.4 Appendix: Interface for the Classes Startup and Measure

The file patop defines.h contains the interface definitions for the classes Measure and Startup and the class and friend methods provided to use these classes.

```
typedef enum {
        DIAGRAM,
        BARGRAPH,
        UNDEFINED
            } Output_type;     // Specifies  the graphical
                               // representation for the measurement

/* Op_list: descriptor for a measurement
   An Op_list can specify one or more performance measurements.
   This structure is used by class Startup to build a description
   record for a measurement. It is used by the class Measure to define the
   specified measurement on the cube. See the description of the classes
   Startup and Measure and the HLTL Documentation for details
*/

struct Op_list {
        Pa_type pa_type;       // Kind of the measurement to be
                               //  (e.g. TASKRUNNING))
        char *title;           // Title of the  measurement
                               // this value is only used, if the
                               // all_in_one component of class
                               // Startup is FALSE
```

```
        char *shorttitle;     // used only for bar diagramms
        char *units;          // units of the measurement (e.g. kB/sec )
        Node *nodes;          // specifies the related nodes for the
                              // measurement
        Output_type output;   // this value is only used, if the all_in_one
                              // component of class Startup is FALSE
        char **objekts;       // specifies the related objects on
                              // the related nodes 'nodes' (optional)
        struct Op_list *next;// next measurement (optional)
               };
/* Token_list: List of the identifiers of all started measurements */

struct Token_list {
         Tool_token id;       // Ident for the measurement
         char name[MAXNAME];  // Name of the measurement  (necessary for
                              // updates in a bargraph
         struct Token_list *next;                 // next element
                  };

class Startup;                // Forward declaration

class Measure {
         // Information required before the measurement is started
        struct Op_list *operations;// Specifier for all the measurements
        Node node;                 // node where the measurement is
                                   // performed
        char* objekt;              // object on which the measurement
                                   // is based

        /* Informations required after the measurement was started */
        int so_far;           // time interval of the measurement
        struct Token_list* id_list; // Identifier of the measurement
        union {                     // Graphical object of the measurement
        Histo    *histo;
        Bargraph    *bargraph;
               };
        XtInterval  Id time_out;
        Boolean interval_started;// specifies if time_out is valid

        public:
        char title[MAXTITLE];           // Title of the measurement
        char* units;                    // units of the measurement (e.g. kB/sec)
```

```
        unsigned long update_interval;  // granularity of the measurement
        int meas_granularity;           // granularity of the diagram
        Output_type output_type;                  // graphical representation
        Measure(Output_type,struct Op_list*,Node,char*,char*,char*);
                  /* Constructor for class Measure
                     defines the measurements but does not start them
                  */
        void ~Measure();                // destructor
        void start();                   // start the measurements defined before
        friend void update(caddr_t);    // update the graphics
        friend void quit_meas(caddr_t); // stop the measurement and delete window
              };
class Startup {
                  /* Class startup is used to build up a list of operations
                     which describes the measurement(s) to be performed
                  */
        struct Op_list *operations;               // Internal representation for
                                                  // a measurement
        struct Op_list* find_entry(Pa_type);
        struct Op_list* new_entry();
        public:
        Boolean all_in_one;   // specifies if all measurements
                              // will be combined in one diagram
        friend void start(Startup&,Node,char*,char*,char*);
                  // create a new instance of class measurement
                  // the number and type of measurements is determined
                  // operations
        Startup();  // Constructor

        void add_operation(Pa_type,char*,char*,char*,Output_type);
        void add_operation(Pa_type,Node*,char*,char*,char*,
                         Output_type);
         void add_operation(Pa_type,char**,char*,char*,char*,
                         Output_type);
                // overloaded method which adds a new initialized
                // element to Op_list
         void remove_operation(Pa_type);
                // counterpart to add_operation
                // removes the first element of Op_list with the right
                // Pa_type tag
               };
```

3 VISTOP

This chapter describes the implementation of the visualization tool VISTOP. In the current state of implementation VISTOP animates communication and synchronization events of a concurrent application. VISTOP visualizes the execution of an application in terms of the basic MMK objects task, mailbox and semaphore. Tasks in a MMK application can communicate via mailboxes and synchronize via mailboxes and semaphores. See the "MMK User's Guide" for a discussion of the MMK programming paradigm. VISTOP's GUI is based on the graphical objects described in part 2 of the design documents. The tool uses some of the HLTL functions (see part 4) in order to get information about the application which is to be animated.

This document consists of two parts. The first part gives some information about the basic implementation ideas and presents VISTOP's module structure. The second part, the appendix, shows the class interfaces of VISTOP.

3.1 The Implementation Idea

VISTOP's basic data structure is a so called "animation queue". An animation queue is a double linked list, which cointains snapshots of the program execution. Each snapshot consists of the status of all observed MMK objects at a certain time. VISTOP consists of two independent parts, the data acquisition part and the data animation part (see figure 3.1). The data acquisition part collects data of the application and stores it in the animation queue, the data animation part reads the contents of the animation queue and produces a sequence of pictures which animate the program execution. The advantage of having two independent parts for acquisition an animation is obvious: you can easily replace one part without affecting the other.

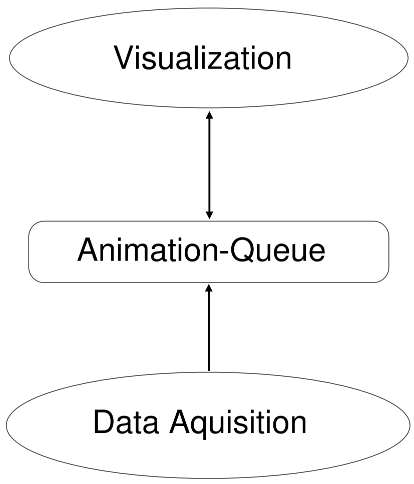


fig.3.1: VISTOP idea

The way how VISTOP animates communication events is described in the "VISTOP User's Guide" and will not be covered here in detail.

3.2 Data Acquisition

Like DETOP and PATOP, also VISTOP uses the HLTL interface for its communication with an application which is running on a remote parallel machine. From the various functions offered by the HLTL layer, VISTOP uses the functions dbreak_s5() and cbreak_s5() to specify or clear breakpoints, and the function inspect_s5() to read the status of the MMK objects of the application. The function go_s5() is used to start the application for the first time and restart it again, after a breakpoint has been reached and the state of the application is read.

In order to collect data of interesting communication events, VISTOP specifies a breakpoint predicate, which depends on the MMK objects selected by the user. The break predicate stops the application, whenever a communication event happens where one of the selected MMK objects is involved. When a breakpoint is reached, a handler function is called, which has to be assigned in the dbreak_s5() function. The handler works in the same way as the handler function used by DETOP for breakpoint notification. When a breakpoint is reached in the application, the handler function is called asynchronously and writes to a pipe which is specified as additional event source for VISTOP (XtAddInput()). This is the only action of the handler. The X Window System calls the function acquire() after an input to the pipe occurred. Acquire() reads the state of all the interesting MMK objects using the HLTL function inspect_s5(). It is necessary to put the pipe inbetween the handler function and aquire(), as you may not perform X Window requests asynchronously. The source code for the data acquisition is located in the file ACQUIRE/Acquisition.c (The path is relativ to the VISTOP root directory see 3.5). This module contains the five functions define_break(), acquire(), init_vistop_break_handler(), vistop_break_handler() and acquisition_enable(). The function define_break() defines a break point specification according to the list of selected objects. This breakpoint causes the application to be stopped, whenever one of those objects communicates. The signal which reports that a break point was reached (SIGUSR1), calls the routine vistop_break_handler() (this is the function named handler in the text before). The break handler causes the function acquire() to be executed. Now acquire() inspects all entries of the list of observed elements and stores the information in a snapshot which is appended to the animation queue.

3.3 VISTOP's Internal Objects and Data Structures

VISTOP uses various different types of objects to perform its animation. These objects and how they interact can be seen in figure 3.2. The central data structure is the animation queue. It consists of a double linked list of Snapshots. Each Snapshot contains the state of the application to be monitored at a certain time. A Snapshot is a list of Snapshot_Elements. The basic component of a Snapshot_Element is the status of one MMK object.

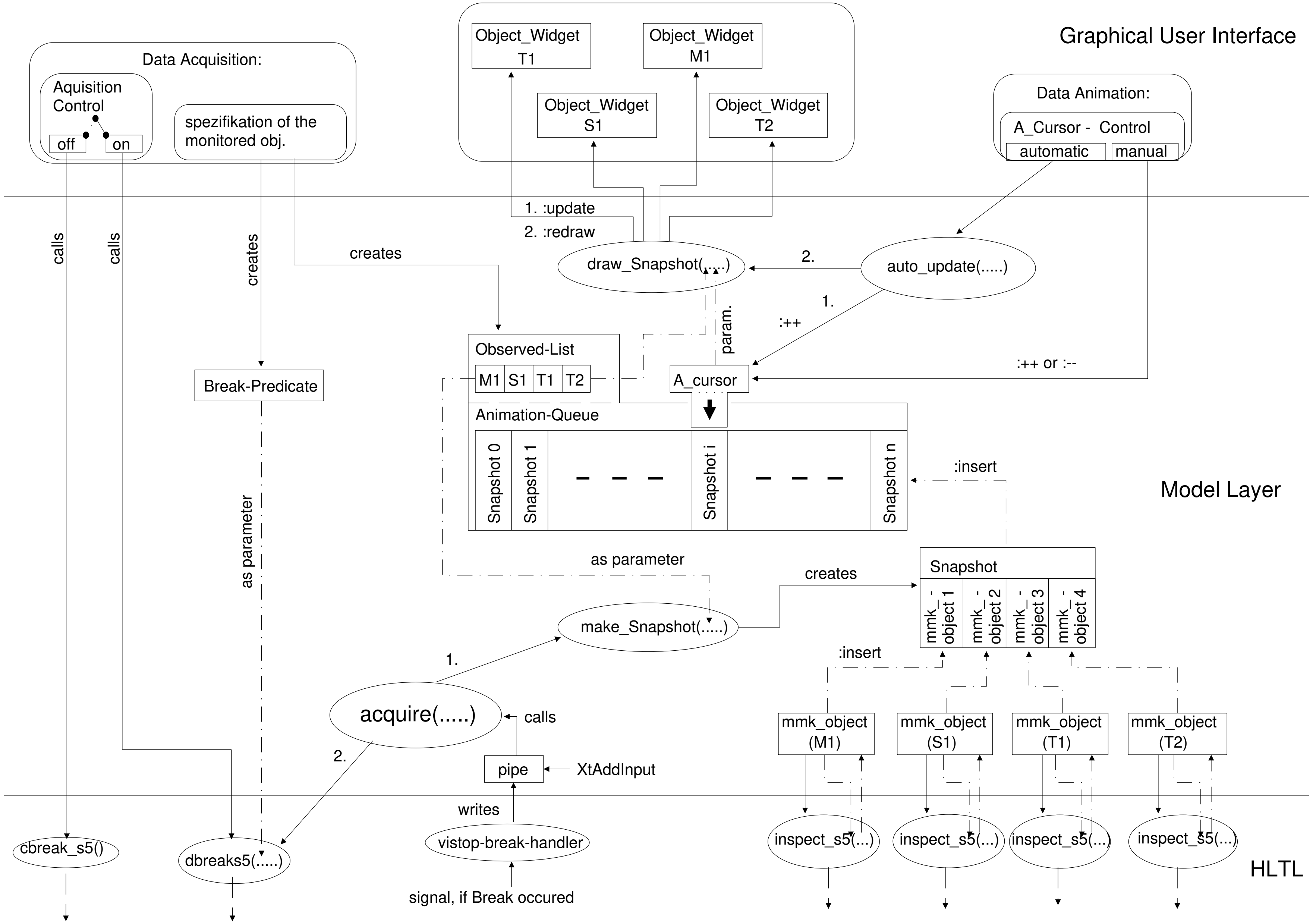
The information which objects of an application are interesting for the user is stored in a so-called obslist. This list contains all objects the user wants to be observed. The most important component of it is a pointer to an Object_Widget, which displays the state of an MMK object. For each of the MMK objects to be visualized there exists one Object_Widget, which displays the status information of the object from a snapshot (see graphical objects).

Note there is only an implicit close relation between a snapshot and the obslist. Each element of the oblist has a corresponding element in each snapshot. Also the sequence of the elements in the obslist and in a snapshot is equal. VISTOP uses this relation to determine, which snapshot_element will be displayed in which graphical object (i.e. the state of the first snapshot_element of a snapshot will be displayed in the graphical object, where the first entry of the obslist points to, etc.).

The box labeled 'Data Acquisistion' and the lower part of the figure 3.2 show how VISTOP creates the animation queue. First a break predicate (dbreak_s5()) is specified and the application is started (go_s5()). When the application reaches the breakpoint, it calls vistop_break_handler(), which causes the function acquire() to be called. The function acquire uses make_Snapshot to read the state of the application, and to store the information into a snapshot and add the snapshot to the animation queue. Then the break predicate is sent again to the application (dbreak()) and the application is restarted (go_s5()).

The rest of figure 3.2 shows, how the data of the animation queue is used for the animation process. VISTOP uses an "animaton cursor" which points to snapshot which is currently displayed in the animation panel.

The animated view of the application is created by the function draw_snapshot(). The animation steps through the stored snapshots either manually or automatically. Manual animation performs one step forward or backward, when the user presses the appropriate button in the animation panel. Automatic steps through the animation queue in a user definable speed.



Graphical User Interface
Data Acquisition:
Aquisition Control
off
on
spezifikation of the monitored obj.
Object_Widget T1
Object_Widget M1
Object_Widget S1
Object_Widget T2
Data Animation:
A_Cursor - Control
automatic
manual
calls
calls
creates
creates
1. :update
2. :redraw
draw_Snapshot(.....)
2.
auto_update(.....)
1.
:++
param.
:++ or :--
Observed-List
M1
S1
T1
T2
A_cursor
Break-Predicate
Animation-Queue
Snapshot 0
Snapshot 1
Snapshot i
Snapshot n
:insert
Model Layer
as parameter
as parameter
creates
Snapshot
mmk_-object 1
mmk_-object 2
mmk_-object 3
mmk_-object 4
make_Snapshot(.....)
1.
:insert
acquire(.....)
calls
2.
pipe
XtAddInput
mmk_object (M1)
mmk_object (S1)
mmk_object (T1)
mmk_object (T2)
writes
cbreak_s5()
dbreaks5(.....)
vistop-break-handler
signal, if Break occured
inspect_s5(...)
inspect_s5(...)
inspect_s5(...)
inspect_s5(...)
HLTL

Fig. 3.2

3.4 Graphical Objects

VISTOP's graphical objects are based on the X Windows Athena Widgets and some additional widgets presented in part 2 of the design documents. Most important for VISTOP are the objects representing MMK objects and the concurrency window. In addition there are some dialog windows that are created, when VISTOP needs information from the user.

- Most of VISTOP's objects are composed of simple widgets. The possibility to do this is inherited from the X Toolkit Intrinsics. The only objects using the higher level objects of part 2 are VISTOP's main menu (uses the MENU class) and VISTOP's error window (uses the MESSAGE class).
- For each of the objects there exists a class interface, which hides the X Window dependencies and offers a comfortable interface to the object.
- The graphical objects which represent the MMK objects have additional methods assigned to iconify or deiconify them and to move them in the animation panel.

MMK objects

The class definition file for VISTOP's graphical objects representing the MMK objects is Object_Widget.h.

This file contains the declarations of the structure Object_Widget and the classes Task, Mailbox and Semaphore, where Object_Widget forms the superclass of the other classes. There are no instances of class Object_Widget in VISTOP. The classes contain methods to create and destroy graphical objects (i.e. Task_Widgets, Mailbox_Widgets and Semaphore_Widgets), to update them (i.e. update the state of the objects according to the contents of a snapshot), to move them etc.

The file Iconify.C defines the callback functions Iconify, DeIconify and Raise_it for the Object_Widgets. They are used to switch between the iconified state (the initial state) and the deiconified state of the Task_Widget, the Mail_Widget and the Semaphore_Widget. The iconified state saves space on the screen but doesn't show all the information of the object. Raise_it puts an object on top of the display stack, if it was hidden before by one of the other objects in the animation panel. These callback functions are assigned to the left mouse button. The raise callback is only working if the object is deiconified and you don't press the mouse button while you are over the icon of the object, which would iconify it. Task_Widgets, Mail_Widgets and Semaphore_Widgets can be iconified using the Icon and Slobject widgets described in part 2 of the design documents.

Fig. 3.3: A Task widget iconfied and deiconfied

Concurrency window

The class definition file for VISTOP's concurrency window is contained in the file A_Widget.h. It contains the structure A_Widget which contains the animation panel for VISTOP (see figure 3.4 for an example). The animation window contains basically a large number of smaller widgets and consists of three parts. The top of the animation widget provides the interface for data acquisition. Here the user can select the MMK objects to be displayed and the user can start and stop data acquisition. The bottom of the panel controls the data animation and the middle part holds all the objects of the animation. The animation widget is used as parent widget for all the Task_Widgets, Mail_Widgets and Sema_Widgets and also as container for all arrows.

The construction method builds up the animation window. It consists of a popup shell, the frame (a form widget) which contains the control areas and the animation panel. It also assigns all the various callback functions the the buttons of the animation window.

Arrows

The module Pfeil.h specifies the object arrow and a list of arrows. Arrows are used to visualize waiting queues before mailboxes and semaphores and message exchange. All arrows used for this process are inserted in a list (class Pfeil_list). This list is used to update the arrows when they disappear from screen because the animation window is obscurred or another object slides over an arrow. The arrows have to be updated explicitely (redraw_list()), because they are not part of a widget and are not updated automatically via the X toolkit intrinsics by an expose event.

The implementation of all graphical objects is straightforward and is omitted.

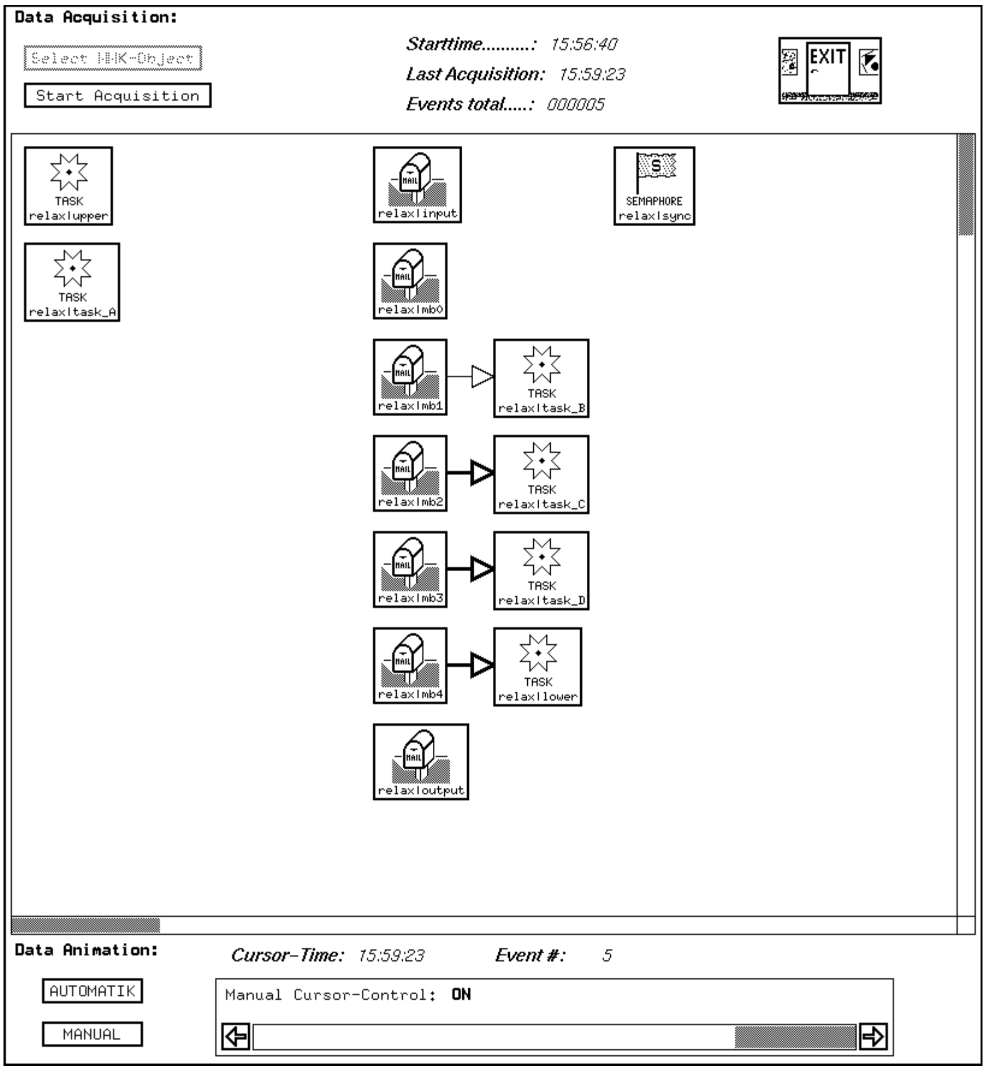


Fig. 3.4: VISTOP animation panel

3.5 Visualization of Communication Events

The visualization of communication events is performed using the graphical objects representing tasks, mailboxes and semaphores and arrows. VISTOP displays the communication events between tasks on the one side and mailboxes respectively semaphores on the other side. The key classes for the animation process are the classes Visual_Queue, Pfeil and Pfeil_list.

Every passive graphical object (i.e. mailbox widget or semaphore widget) has its own Visual_Queue (Task widgets don't have a visual queue).

The methods of the the class Visual_Queue can insert tasks in a queue in front of a mailbox or semaphore, when a task communicates with a passive object. Communication means either non blocking communication (sending,receiving) or blocking communication (waiting to send or waiting to receive). Blocking and non blocking communication can be distincted by the arrow widths between the objects.

When a task is inserted in a communication queue it is moved in front of the mailbox or semaphore (if there is enough space available) and an arrow is drawn between the task and the last object waiting before the object the task is communicating with. The arrow is inserted in an arrow list (see Pfeil.h). When a task is removed of the queue the arrow is deleted from the arrow list and the task goes back to its old position also called home position. Additionally the class Visual_Queue is responsible to update all arrows concerned by the insertion or removal of a task.

3.6 Module Structure of VISTOP

VISTOP is mostly implemented in C++ and is physically divided in the interfaces of the modules and the source code which implements the various methods. The code for VISTOP is located in the tree beyond the subdirectory /usr/proj/ipsc2/TOPSYS/HOST/TOOLS/VISTOP. However the location of the root directory for VISTOP is not mandatory.

Logically VISTOP consists of two parts, the data acquisition part and the data animation part. The data acquisition part which is much smaller than the animation part has the task to gather run-time information of the application to be visualized and store it in the animation queue. The data animation part forms the user interface for the animation process. The code for the class methods itself rests in the 4 subdirectories ACQUIRE, ANIMATE, OBJECTS and UTILITY. All header files reside in the subdirectory INCLUDE/CC. The various bitmaps VISTOP uses for its graphical objects are located in INCLUDE/CC/bitmap.

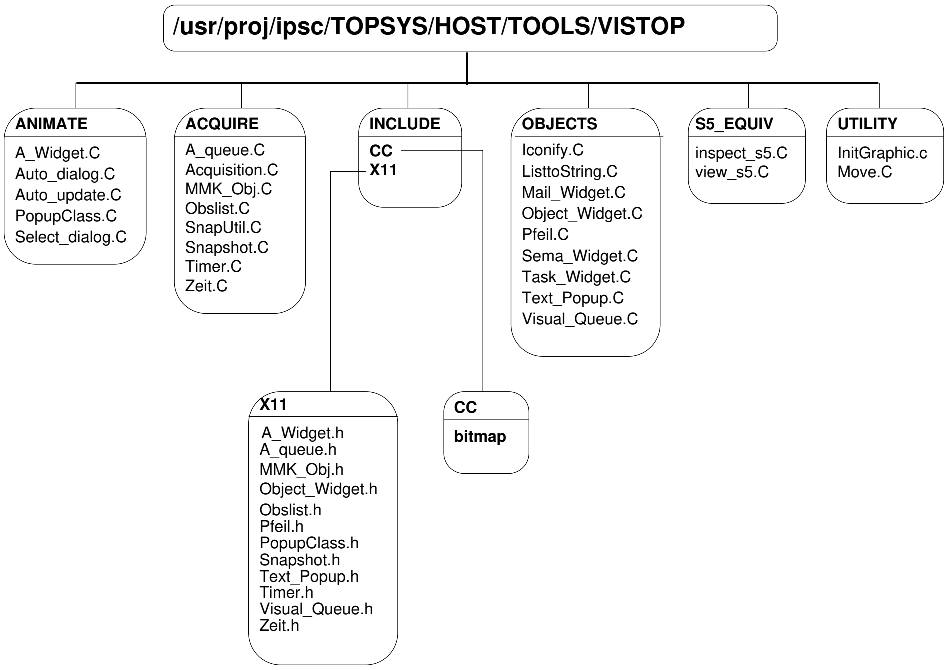


Fig. 3. 5: VISTOP module structure

Following is a short guide through the modules that form VISTOP's source code, which have no class interface associated and the classes that are not contained in the appendix. This part gives a short hint what all of VISTOP's modules are supposed to do. However this section will not give you much deeper insight into VISTOP basic principles and is basically intended for those users who also plan to refer to VISTOP's source code.

Auto_dialog.C

Implementation of the AUTOMATIC - Button of the animation widget A_Widget. It creates a pop up widget to determine the update frequency for the animation (see figure 3.6).

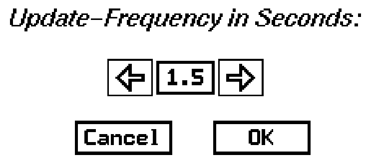


Fig. 3.6: Auto dialog window

Auto_update.C

Implements the automatic update function of the animation. This function will be called by a programmable timer when the user selects automatic mode for the animation process.

Select_dialog.C

Implementation of the Select MMK-Object - Button of the animation widget A_Widget. Creates a pop up window to select the MMK-objects to be visualized (see figure 3.7)

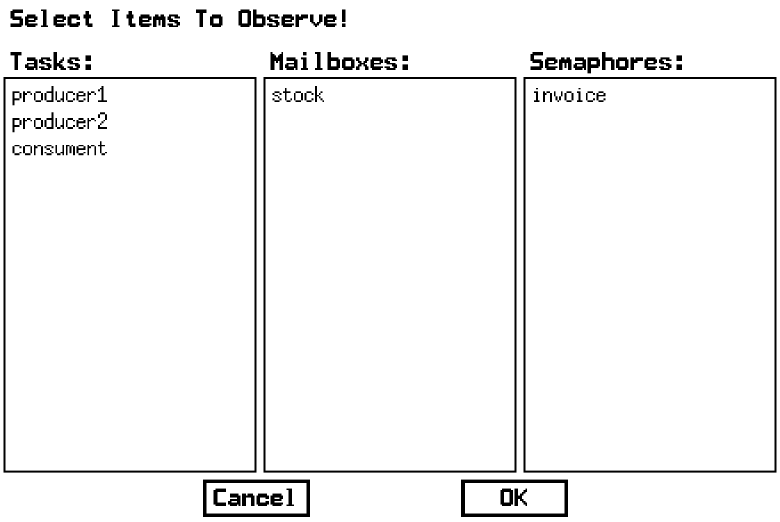


Fig. 3.7: Select MMK-Objekt window

SnapUtil.C

Contains the two functions make_Snapshot and draw_Snapshot. These functions are very important for VISTOP's animation. The function make_Snapshot() collects data of an application and stores it in a snaphot. This information is used by draw_Snapshot(). draw_Snapshot() compares the state of the objects in the snapshot with the state of the objects currently shown on the Animation panel and moves the graphical objects into their new state. To do so it uses the class Visual_Queue.

Timer.h

Defines the class Timer. This class can be used to assign a TimerCallbackProcedure to an X window client. It is used for the automatic mode of the animation. Before you can use a timer you first have to create it and afterwards set it.

The class timer is implemented in the file Timer.C

Zeit.h

Defines the structure Zeit which which gives an interface to the system time in the form HH:MM:SS.

The methods for structure time are implemented in Zeit.h

InitGraphic.c

Implements the functions to initialize the graphical objects.

Init_Schirm:	init the Screen
Init_Cursors:	set shapes of the cursors
Free_Cursors:	free the cursors created in Init_Cursors
Init_Pixmaps:	set all the pixmaps for VisTop
Free_Pixmaps:	free all the pixmaps created in Init_Pixmaps
Init_Fonts:	set all fonts for VisTop
Free_Fonts:	free the fonts

Move.C

Implements the function Move_Handler. Move_Handler is a callback function. It can be assigned to a SLObject- or an Icon-Widget which (see the description of the SLObject widget and the Icon widget in part 2 of the design documents) represents a MMK-object. Move_Handler moves the object according to the mouse motion. VISTOP uses this function for moving tasks, mailboxes and semaphores in the animation window.

Main.C

The file Main.C contains the main loop of VISTOP. This file is located in the VISTOP root directory. Main initializes the creates the VISTOP command menu and adds the callback functions for the concurrency and exit entry.

3.7 Appendix : Interface definitions of VISTOP

The module interfaces are written in C++ and also presented in C++ syntax within this document.

Object_Widget.h

This file contains the declarations of the structure Object_Widget and the classes Task, Mailbox and Semaphore. These are the graphical objects representing MMK objects. The implementations for the classes is contained in the files Object_Widget.C, Task_Widget.C, Mail_Widget.C and Sema_Widget.C.

```
struct Object_Widget {
    Widget widget;              // widget shown when deiconified
    Widget icon;                // Icon widget, shown when iconified
    Boolean iconified;          // TRUE if Widget is iconified;
    Boolean needs_redraw;       // TRUE if Widget is iconified;

    Pfeil *P_links;             // Arrow to visualize relation
                                // between objects in queues
    Richtung R_links;           // Direction of the left arrow
    Pfeil *P_rechts;            // Arrow to the right
    Richtung R_rechts;                              // Direction of the arrow

    virtual void move(Position, Position);          // move object
    virtual void slide(Position, Position);         // animate motion
                    // return position and size of the object
    void geometry(Position&, Position&, Dimension&, Dimension&);

    virtual void update(Obj_Desc*, obslist*);
    virtual void redraw();

    virtual void iconify();
    virtual void deiconify();
```

```
    Object_Widget() {};
    ~Object_Widget()
    {
                    XtDestroyWidget(widget);
                    XtDestroyWidget(icon);
   }
};

class Task_Widget : public Object_Widget {
    Widget state;    // subwidget for the state of the task
    Widget wait1;    // subwidget Text: Waiting at .. (line 1)
    Widget wait2;    // subwidget Text: to do something (line 2)
    Task_Desc* desc; // task-description (state of the task)

    void draw(Widget, Position, Position);          // draw widget for the
                                                    // first time
 public:
    Task_Widget(Obj_Desc*, Widget, Position, Position);
    void update(Obj_Desc*, obslist*);
    void redraw();
};

class Sema_Widget : public Object_Widget {
    Widget units;              // subwidget for the number of units
    Widget waitbutton;         // subwidget to show if tasks are waiting
    Text_Popup *tasklist;      // subwidget for the tasks waiting
    String tasklist_text;      // names of the waiting tasks
    Sema_Desc* desc;           // sema-descriptor

    void draw(Widget, Position, Position);          // draw widget for the
                                                    // first time

public:
    Sema_Widget(Obj_Desc*, Widget, Position, Position);

    void update(Obj_Desc*, obslist*);
    void redraw();

    void iconify();
    void move(Position, Position);

    Visual_Queue *queue;       // queue for the waiting tasks
```

```
    ~Sema_Widget()
    {   delete queue;  };
 };
class Mail_Widget : public Object_Widget {
    Widget bar;              // subwidget for the contents of the mailbox

    Widget contbutton;       // contents / no contents
    Text_Popup *inhalt;      // message pointer of the mailbox
    String inhalt_text;

    Widget waitbutton;       // waiting tasks
    Text_Popup *tasklist;
    String tasklist_text;

    void draw(Widget, Position, Position);

    Mail_Desc* desc;

public:
    Mail_Widget(Obj_Desc*, Widget, Position, Position);

    void update(Obj_Desc*, obslist*);
    void redraw();

    void iconify();
    void move(Position, Position);

    Visual_Queue *queue;     // queue for the waiting tasks
    ~Mail_Widget()
    {    delete queue;   };
 };
```

Pfeil.h

The module Pfeil.h specifies the object arrow and an list of arrows. Arrows are used to visualize waiting queues before mail_boxes and semaphores.

```
class Pfeil {
    Widget widget;           // Widget, where arrow is drawn

    XPoint punkt[6];         // Array of points of the arrow
    double l1, l2;
```

```
    Boolean drawn;

    GC gc;
   GC kill_gc;

public:
    Pfeil(Widget, int, XPoint&, XPoint&);          // draw arrow the first time
    // Parameter :
    //   Widget:  where shall arrow be drawn
    //   int:     width of the arrow
    //   XPoint&: begin of the arrow
    //   XPoint&: end of the arrow

    ~Pfeil();       // remove arrow

    void draw(XPoint&, XPoint&);

    Boolean is_drawn() { return drawn; };

    void redraw();
    void erase();

    void geometry(XPoint&, XPoint&);  //  return coordinates of the arrow
};

struct Pfeil_element {
    // Pfeil_element forms a list data type for Pfeil_list
    friend class Pfeil_list;

    Pfeil *pfeil;
    Pfeil_element *next;

    Pfeil_element( Pfeil*, Pfeil_element* );       // insert arrow in list
};


class Pfeil_list {
    //Pfeil_list contains all arrows of the animation window
    Pfeil_element *anker;

public:
```

```
    Pfeil_list();              // new empty Pfeil_list
    Pfeil_list( Pfeil* );      // new Pfeil_list

    void insert( Pfeil* );     // insert arrow at the begin of the list

    void remove( Pfeil* );     // remove arrow from list
    void remove_all();                             // remove Pfeil_list

    Boolean is_empty();                            // check if Pfeil_list empty

    void redraw_list();                            // redraw all arrows

     ~Pfeil_list()    {     remove_all();   } };
```

The implementation of the interface can be found in Pfeil.C

Visual_Queue.h

Visual_Queues are used to visualize communication relations between MMK objects. The implementation for Visual_Queue.h can be found in the file Visual_Queue.C.

```
class Visual_Queue {
    visual_element *anker, *ende    // Pointer to the first and last element
                                    // of the visual queue
    Object_Widget *vater;          // mailbox or semaphore where
                                   // tasks are waiting

    Zustand queue_state;       // state of the queue

    Boolean check_position(Object_Widget*, Object_Widget*, Zustand);
                    // insert task in Visual_Queue
                    // Parameter:
                    // 1. last widget of the queue or vater
                    // 2. task to be inserted

    void rm_widget(Object_Widget*, Object_Widget*, Object_Widget*);
                    // remove task from Visual_Queue
                    // Parameter;
                    // 1. Predecessor in the queue
                    // 2. task to be removed
                    // 3. Successor in the queue
public:
    Q_Lage lage;               // are the tasks waiting at the right or left
```

```
                          // side
    Visual_Queue(Object_Widget*, Q_Lage);
                 // Parameter:
                 // 1. The Widget that gets a waiting queue
                 // 2. waiting at the left or at the right side

   Boolean insert(Task_Widget*, Zustand);
                 // check, if there is enough space in the animation widget
                 // to insert a waiting task in the queue

   void remove(Task_Widget*);
                 //  remove Task from queue
   void remove_all();
                 // Remove the whole queue
   visual_element *is_element(Task_Widget&);
                 // check if the queue contains task as element
                 // return pointer to the task if ok
                 // else NULL
   Boolean is_empty();       // check if queue is empty
   ~Visual_Queue()
  {
                 remove_all();
   }
};
```

A_Widget.h

The structure A_Widget contains the definition of the animation panel for VISTOP. The implementation for A_widget.h can be found in A_Widget.C

```
struct A_Widget {
    // Generates and draws the Animation window
     A_Widget(Widget);
     ~A_Widget();

    Widget popup_shell;  // Shell for the animation window
    Widget self;

    Widget sel_obj;      // pointer to the select MMK window
     // MMK objects which are displayed in the animation panel
    Widget task_list;
    Widget sema_list;
    Widget mail_list;
```

```
    char **break_praed;         // Break predicate used to call the break
                                // function
    Widget st_acqu;       // Button showing Start or Stop Acquisition
    Boolean acqu_running; // True, if Acquisition is running

    Widget st_time;      // widget showing start time of the acquisition
    Zeit start_zeit;
    Widget now_time;     // widget showing time of the last acquistion

    Zeit jetzt_zeit;  // Time now
    Widget eve_tot;      // widget showing number of snapshots


    Widget desktop;     // Animation panel

    //  Widgets of the animation control area

    Widget auto_enable;        // widget with label 'AUTOMATIK'
    Widget manuell;            // frame around manual cursor control
    Widget man_switch;         // widget with label 'MANUAL'
    Widget bar;                // scrollbar widget
    unsigned long scroll_pos;  // position of the scroll thumb
    Widget step_left;          // widget showing the left arrow
    Widget step_right;         // widget showing the right arrow
    Widget csr_t;              // widget showing time of the snapshot in the
                               // animation panel
    Widget ev_nr;              // number of the snapshot in the panel

     // Timer which performs the automatic animation
    Timer *auto_timer;
    obslist *observed_elements; // list of MMK objects which are observed
    A_cursor *Animation_Cursor; // Current position in the animation queue
};
```

A_queue.h

This file contains the declaration of the animation queue (class A_queue). This is the toplevel data structure for VISTOP and holds the data collected from the application running. The animation queue is a double linked list of Snapshots (see Snapshot.h).

A_queue.h contains three class definitions:

A_element: Insert and remove one Snapshot in a double linked list

A_queue: Builds a double linked list of A_elements.

A_cursor: Builds a pointer to one of the Snapshots of the animation queue. The animation cursor always points to the Snapshot that is currently shown in the animation window.

The implementation of the file can be found in A_queue.h

```
class A_element {
friend class A_queue;
friend class A_cursor;

        A_element* prev;        // Predecessor
        A_element* next;        // Successor
        Snapshot* cont;
        A_element(Snapshot*, A_element*, A_element*);
        ~A_element();
 };

class A_queue {

friend class A_cursor;
        A_element* anf;
        A_element* ende;

          unsigned long max;

public:
        A_queue();
        ~A_queue();

        void insert(Snapshot*);  // append Snapshot at the end of the queue
        void rm_first();         // remove first element of the queue
        void clear();            // remove the whole queue
        unsigned long length()   // return length of the queue
};

class A_cursor {
        A_queue*    queue;     // pointer to the animation queue
        A_element*  cursor;    // pointer to the current element



public


        A_cursor(A_queue&);    // put cursor to the beginning of the
```

```
                                animation queue
        Snapshot* home();      // return first snapshot
        Snapshot* end();       // return last snapshot

        Boolean is_at_home();  // return TRUE, if cursor is at home position
        Boolean is_at_end();   // return TRUE, if cursor is at end  position

        Snapshot* next();
        Snapshot* operator++() // move cursor to the next position

        Snapshot* prev();
        Snapshot* operator--()  // move cursor to the previous position

        Snapshot* aktuell()    // return the Snapshot the cursor points to.
};
```

Snapshot.h

A Snapshot is a linear list of Snapshot_Elements. A Snapshot is created, whenever an interesting event occurs in the application which is visualized. Each Snapshot_Element holds the status information of one MMK object (i.e. task, mailbox, and semaphore). Snapshot.h contains the methods to manipulate such an object of class Snapshot.

Snapshot.C implements the methods of Snapshot.h

```
struct Snapshot_element {      // Type definition for a snapshot
    mmk_object *mmkobj;

    Snapshot_element *next;

    Snapshot_element(mmk_object*, Snapshot_element*);// constructor
    ~Snapshot_element();                            // destructor };

class Snapshot {               // Class definition for a snapshot
    friend class Snapshot_iterator;

    Snapshot_element *anker;   // Pointer to the first element of the list
    Snapshot_element *ende;    // Pointer to the last element
public:
    Zeit acqui_zeit;           // acquisition time
    unsigned long count;       // snapshot number

    Snapshot();
```

```
    ~Snapshot();

    void insert( mmk_object* );  // Append an object at the end
    void remove( mmk_object* );  // Remove an object
    void remove_all();           // Remove all objects of the snapshot
};

class Snapshot_iterator {      // Iterator through a snapshot

    Snapshot                   *list;
    Snapshot_element           *aktuell;

public:
    Snapshot_iterator( Snapshot& );               // point to first object of the
                                                  // snapshot
    Snapshot_element *operator()();               // return aktuell
                                                  // point to the next object
};
```

MMK_Obj.h

Defines the structure mmk_object which is a super-class of the three different MMK-objects Task, Mailbox and Semaphore. Also the class definitions for Task, Mailbox and Semaphore are included. The constructor of the classes Task, Mailbox and Semaphore initializes the created objects by calling the HLTL routine inspect_S5 (refer to the HLTL documents for further details), which reads the status of the selected object of the application. Instances of class mmk_objects are used to build up a snapshot.

The destructor and inspect methods are implemented in the file MMK_Obj.C

Interface:

```
struct mmk_object {
    Obj_Desc desc;    // this entry holds the status information for
                      // one MMK object (i.e it is a union which is
                      // described in part 6 of the design documents.

    virtual Errmsg *inspect() { return NULL; };
};

class Task : mmk_object {
    void delete_taskdesc();
```

```
public:
    Task( Errmsg *error, char *name )
    {
                    inspect_s5(error, T_task, name, &desc);
    }
    ~Task()
    {  delete_taskdesc();  }
    Errmsg *inspect();
};

class Mailbox : mmk_object {
    void delete_maildesc();

public:
    Mailbox( Errmsg *error, char *name )
    {
                     inspect_s5(error, T_mail, name, &desc);
    }

    ~Mailbox()
    {  delete_maildesc();  }

    Errmsg *inspect();
};

class Semaphore : mmk_object {
    void delete_semadesc();

public:
    Semaphore( Errmsg* error, char* name )
    {
                    inspect_s5(error, T_sema, name, &desc);
    }
    ~Semaphore()
    { delete_semadesc(); }
    Errmsg *inspect();
};
```

Obslist.h

The class obslist (list of observed objects) is used to hold the list of mmk-objects which the user wants to be used for visualization. The basic elements of an obslist are of the type obs_element. The implementation of the methods for the file obslist.h can be found in obslist.c

```
struct obs_element {
    Type_tag type;            //  type of mmk-objects t_task, t_mail, t_sema
    char* name;               //  name of the object as string
    Object_Widget *ow;        //  widget that represets the object

    obs_element *next;

    obs_element( type_tag, char*, obs_element*, object_widget* );
    ~obs_element();

    void print();
};

class Obslist {
    friend class obslist_iterator;

    obs_element *anker;

public:
    obslist();                                        // anker = NULL
     obslist( Type_tag, char*, Object_Widget* );      // create obslist
                                                      // consisting only of
                                                      // Object_Widget

    void insert( Type_tag, char*, Object_Widget* );
                    // insert Object_Widget at the beginning of obslist
    void remove( char* );     // remove first object with the name
                              // of the call parameter
    void remove_all();        // remove all Objects
    void print();             // dump the list of the observed elements

     Boolean is_empty();      // returns TRUE if anker == NULL
     Object_Widget *is_element(  char*  );            // returns objects if
                                                      //  it is in obslist

     A_queue Animation_Data;

     ~obslist()
};

class obslist_iterator  {
```

```
     obslist          *list;
     obs_element      *aktuell;

public:
     obslist_iterator( obslist& );                    // Point to first element

     obs_element *operator()();                       // go to next elemement
};
```

4 TOPCOM

This chapter contains the common code for PATOP and DETOP. At the moment, this consists only of the error messages. The directory TOPCOM is located parallel to the directories DETOP, PATOP and VISTOP in /usr/proj/ipsc2/TOPSYS/HOST/TOOLS/TOPCOM.

errmsgs.C

This file contains the error messages for DETOP and PATOP and the functions to show error messages. It creates and initializes the error window and contains the functions to display error messages. Out of historical reasons DETOP and PATOP use different functions to display there Errors. DETOP uses the function dtarget_error(). It displays an error message consisting of two parts. The first part is determined by the error number, the second part is a string which will be displayed. PATOP uses the function error_window(). Internally it uses the function dtarget_error().

Part 4:
The Universal Tool Layer UTOL

Roland Wismüller

This part of the TOPSYS design document is a description of the UTOL, a software layer that is responsible for the transformations between different abstraction levels and for managing the distributed monitoring system. The first chapter gives a short overview of the UTOL, the next two chapters describe the two sublayers HLTL and UDTL in more detail. The appendix, finally, contains a full description of the interface between the UTOL and the tools.

1 Overview

The Universal Tool Layer is the connection link between the tools for parallel systems on one side and the distributed monitoring system on the other side. In the TOPSYS hierarchical tool model (see part 1) this layer is shown as 'Predicate/Action/Symbol Management'. It consists of a C-library that is linked with any tool that wants to use the tool interface to get information about the dynamic behavior of a parallel program. This interface offers a variety of useful functions such as start tasks, set breakpoints, display and modify variables, view task state, trace functions, performance traces and a lot of others (the appendix contains a detailed interface description). All these functions work on the highlevel language (HLL) abstraction level. This means that the objects that are passed to these functions are the objects the application programmer deals with:

- Names of MMK objects (task names, semaphore names and mailbox names)
- Names of program modules, procedures, variables and types
- HLL statements and expressions
- Line numbers in the source code

The UTOL transforms these objects into objects that are expected by the distributed monitoring system:

- Tokens of MMK objects
- Addresses of machine statements, data and tokens
- The number of the processor node that contains the object

The results generated by the monitors (for instance traces) are transformed in the reverse direction and are presented at the tool interface in a HLL oriented way. So the UTOL mainly has to do a mapping from names to addresses, tokens and nodes and vice-versa. In addition, for the monitoring of global predicates, which is necessary to detect breakpoints, the UTOL has to program the communication between the distributed monitors.

In order to keep the UTOL portable and flexible, we have subdivided it into two sublayers. One of them, the Highlevel Tool Layer (HLTL), depends on the specific HLL that the tool interface has to support, the other, the Universal Debug Tool Layer (UDTL) is independent of the HLL. Of course the UDTL cannot support every programming language you can think of, but it is general enough to work with all imperative block oriented programming languages, such as C, FORTRAN, Pascal, Modula2 etc. Because the HLTL is much smaller than the UDTL it is relatively easy to change the UTOL in order to support other HLLs.

In the next two paragraphs and the appendix we describe the operation and implementation of the UTOL sublayers in more detail and provide a detailed description of the interface offered by the HLTL.

2 The Highlevel Tool Layer (HLTL)

2.1 Overview

The HLTL is the highlevel-language-dependent part of the UTOL. It translates command descriptions passed to the tool interface functions that are related to a specific programming language into descriptions that are nearly at the same level of abstraction but are independent of the programming language.

Two classes of objects make the command description of the HLTL interface language-dependent: The first class consists of expressions and statements. The second class are the data types used by the programming language. There are large differences between the data types allowed by different languages, so we regard the types as being HLL dependent. The task of the HLTL is therefore to parse and evaluate expressions and statements, to split complex data types into simple ones and to combine a value of a complex type from its simple component values.

Beside this, the HLTL performs an error checking on all parameters passed to the tool interface. This means that all objects are checked for existence and correct type. For historical reasons (the HLTL interface was a command interface in early versions of the UTOL) most of the parameters are ASCII strings and therefore have to be parsed and translated into data structures. This is also accomplished by the HLTL.

2.2 The Modules of the HLTL

The HLTL can be divided into eight major modules that are shown in figure 2.1. We will briefly describe the function of each module in this chapter. The actual structure of the source code, however, does not perfectly match this figure because the source code is organized in a more command oriented way.

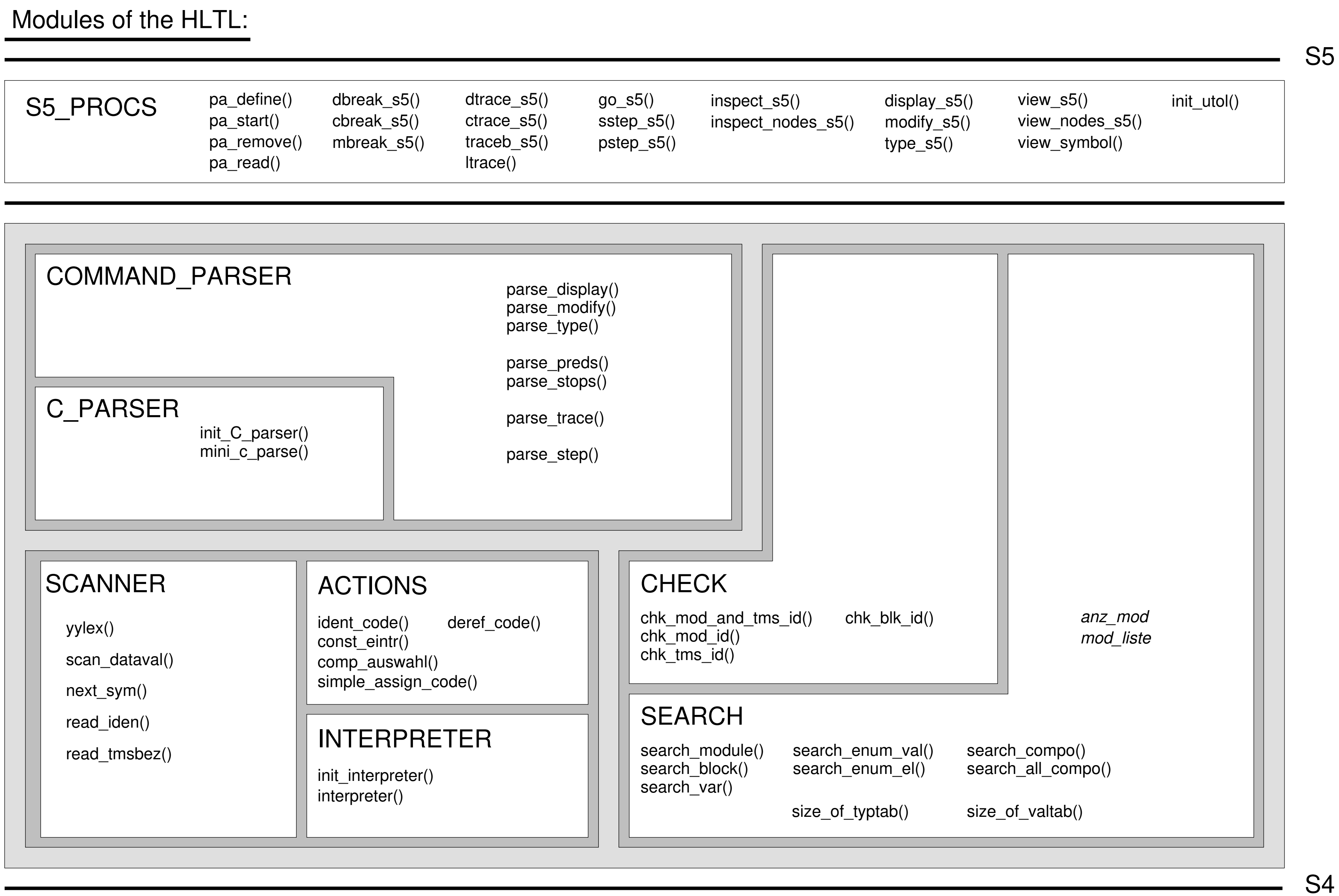


Fig. 2.1: Modules of the HLTL

The Module S5_PROCS contains all procedures that form the TOPSYS tool interface (see appendix for details). They fall into six categories:

- Breakpoint and trace commands:

  dbreak_s5, cbreak_s5, mbreak_s5, dtrace_s5, ctrace_s5, traceb_s5, ltrace

  They are used to define or clear a breakpoint in the application program and to define, clear and view various kinds of traces.
- Go and single step commands:

  go_s5, sstep_s5, pstep_s5

  These commands start tasks or execute a task in a single step mode.
- Inspect and view commands:

  inspect_s5, inspect_node_s5, view_s5, view_nodes_s5, view_symbol

  This set of procedures retrieves state and symbol information about the application program, for instance task state, list of all tasks, mapping and so on.
- Examination of variables:

  display_s5, modify_s5, type_s5

  These commands can be used to read or write variables, to evaluate expressions in the current context of the application and to determine the type of a variable. The type command is completely handled by the HLTL.
- Performance measurements:

  pa_define, pa_read, pa_remove, pa_start

  With these procedures it is possible to define, read, clear and start performance measurements of the application, e.g idle times or queue lengths.
- Initialization:

  init_utol

  This is a procedure used to initialize the HLTL and the UDTL. It builds all global data structures needed by these layers.

The parsers for the arguments of the tool interface procedures can be condensed into a single module COMMAND_PARSERS. The parsers for the display, modify and type commands are heavily based on the C-parser, while the parsers for the breakpoint and trace predicates use only a small part of the C-parser to examine variable descriptions. They are implemented either as simple recursive descent parsers or with the help of the compiler generator yacc for the more complex ones. The main work done by the command parsers is to perform syntactical analysis on the input strings, to check all object names within the string for existence and appropriate type and to build data structures from the parsed information, that are passed to the UDTL.

The C_PARSER module is in many ways the most important module of the HLTL. It is responsible for the handling of C-expressions and C-statements (at the moment only assignments are implemented) and is therefore the part of the HLTL that is most dependent on the supported programming language. In order to simplify the porting to another language, the C-parser is implemented using the compiler generator yacc. It works together with an interpreter module that uses a stack of *Code_el* structures as its main data structure. The stack is a simple array with two functions provided for push and pop operations. The *Code_el* structure itself is a description of a subexpression's value that looks like this:

typedef struct code_el {

BOOL L_value;	TRUE, if the *Code_el* structure describes an L-value, FALSE otherwise.
int op_type;	VARIABLE: *Code_el* describes a C-variable. The structure describes either an access path to that variable or its value. CON: *Code_el* is a constant or the result of an expression. In this case the *id*, *global*, *plist_anf*, *plist_end* and *offset* components are unused.
char *id;	Pointer to the variable's name.
BOOL global;	TRUE, if global variable, FALSE otherwise.
int type;	The type of the *Code_el* as in the type table (primitive type, array, structure and so on).
int type_ind;	The type table index of the type.
Path_list *plist_anf, *plist_end;	Pointer to the beginning and the end of a *Path_list* structure describing the access path to the variable.
int offset;	Byte offset from the base address of the variable. Used for arrays and structures.
struct code_el *left, *right;	Unused.
Simple_type value;	The value of the variable, constant or expression.

} Code_el;

The C-parser can work in three different operating modes: In 'modify mode' it supports the largest grammar, including assignments, but returns no result. In 'display mode' only expressions are accepted and a pointer to the topmost interpreter stack element is returned. The 'type mode' works almost identically to the 'display mode', but only the type of an expression is determined rather than its value. The operating mode is fixed when the C-parser is initialized by a call to *init_C_parser()*. Beside this, the evaluation context, i.e. the scope of variables, and the locations for the return values are specified. The *init_C_parser()* routine simply copies these parameters to some global variables, because the C-parser itself cannot have any parameters due to its implementation using yacc. The interface procedure of the C-parser, *mini_c_parse()*, takes a pointer to the input string as its argument, copies it to a global variable and invokes the real parser. As already mentioned, the return value of the C-parser is a pointer to a *Code_el* structure. If this structure describes a constant, it contains the value of the constant, but if it describes a variable, it contains only the access path to the variable and a type description. The actual variable access is not performed by the parser but is managed by the caller.

Whenever the parser has to perform a semantical action associated with a rule of the grammar, it calls a procedure of the ACTIONS module. There are essentially three kinds of different actions:

First, there are a lot of procedures performing some simple semantical checking, e.g. type checking and/or conversion and checking the existence of names.

Another group of actions generate a new *Code_el* structure for variables or constants on top of the interpreter stack. This includes searching for the identifier, determining the access path and the type of variables and the initialization of the new stack element. These routines are called whenever the parser detects a variable name or a constant in its input string.

The last group consists of interface procedures to the interpreter module. The interpreter is called when the C-parser detects that a variable must be read or written or that an arithmetic or logic operation has to be performed. The interpreter is implemented as a stack machine, that takes the topmost *Code_el* elements from the stack, performs the requested operation and pushes the resulting *Code_el* structure onto the stack. This strategy is best suited for evaluation of expressions. The operations, including assignments, are all performed directly by the interpreter without interaction with the monitors. This evaluation does not include any load or store operations. The parser must invoke the interpreter with a GET_CONT or PUT_CONT command, when it detects that a value must be loaded or stored to a variable. The interpreter then builds an access path to the variable by concatenating the *path_list* (see chapter 3.8 for a description) passed to *init_C_parser* with the *path_list* from the topmost stack element that is maintained by the interpreter. The resulting access path is then passed to the UDTL procedure *display()* or *modify()*, respectively, to perform the load/store operation. Because the C-parser has no need to load/store complex values (e.g. structures) as a whole, the interpreter only handles simple data types. The handling of complex data types in the HLTL is described in chapter 2.4.

The command parsers and the C-parser use various scanners to detect symbols, identifiers, constants or MMK object names in their input strings. In figure 2.1 the scanners are comprised a single module SCANNERS.

For the various checks for existence of names the HLTL contains a few functions that check module, block and MMK object names for existence and correct type. These functions are based on the access functions for the HLTL symbol table, that are used to search for the following names:

- Module names (names of source modules)
- Block names (names of procedures or functions)
- Variable names
- Type names
- Component names of enum types
- Component names of struct/union types

The search algorithm is a hierarchical linear search in a linked structure of arrays that forms the HLTL symbol table (see next chapter).

2.3 Symbol Management in the HLTL

2.3.1 Structure of the HLTL Symbol Table

The HLTL has its own symbol table that contains all symbol information needed by this layer. This includes information about the modules of the application program, the blocks (functions and procedures) within the modules, the variables and the data types. It does not include addresses or storage information of variables. The addresses and the mapping information is handled in the UDTL.

The symbol table is accessed via two global variables: *mod_liste* is an array of module descriptors, *anz_mod* contains the number of entries in this array.

Each module descriptor in the array describes the symbols of one source module. This description consists of the module name, a pointer to the block table, the number of blocks, a pointer to the variable table, the number of variables, a pointer to the type table and the number of types in that module (see fig. 2.2).

mod_name	Name of the source module
blks	Pointer to the block table for this module
anz_blocks	Length of the table
vars	Pointer to the variable table for this module
anz_vars	Length of the table
types	Pointer to the type table for this module
anz_types	Length of the table
...	Source code information, unused

Fig. 2.2

Global names (i.e. names known to every module) are supported by introducing a dummy module named "_global_" in the module table. All global names are then treated as if they belong to the the special module "_global_". In addition, the block table for the "_global_" module contains a special entry named "typedef" that contains all types that are created by typedef statements.

The block table contains an element of type *Blockel* for each block within a module. This element contains the name of the block, the index of the block's first variable entry in the variable table, the number of variables, the index of the block's first entry in the type table and the number of types:

blk_name	Name of the block
var_index	Index of first entry in the variable table for this block
var_anz	Number of variables in this block
typ_index	Index of first entry in the type table for this block
typ_anz	Number of types in this block

Fig. 2.3

Again we have introduced a dummy block with the name "" (i.e. empty string) that is used to access all global static names (i.e. names that are valid in all blocks of one given module).

The variable table is an array of pairs built up by the variable's name and its type represented by an index into the type table:

vname	Name of the variable
vtype	Index of type header in the type table

Fig. 2.4

The type table is the most complex of the HLTL symbol tables. It is really a tree structure, corresponding to the hierarchical definition structure of types, that is flattened and stored in an array. Each table entry has the following form:

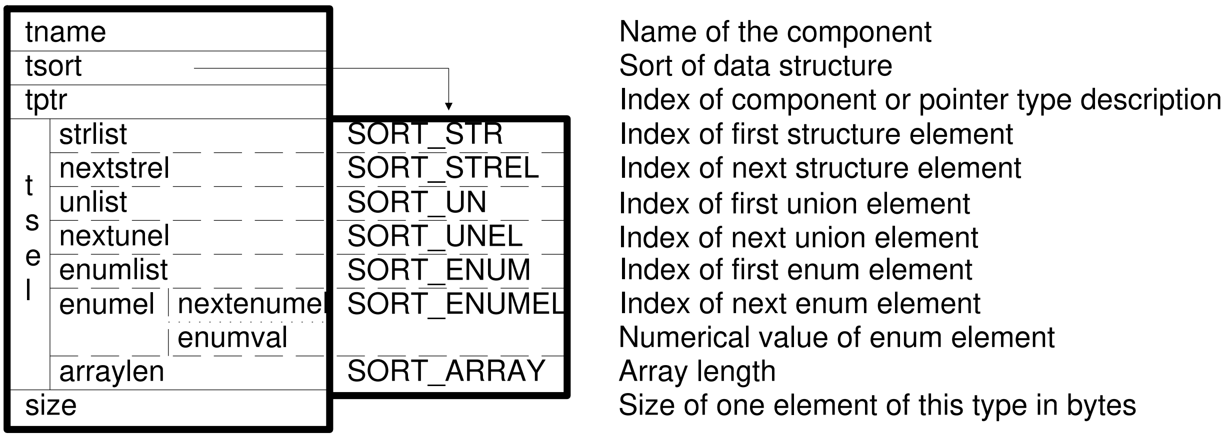
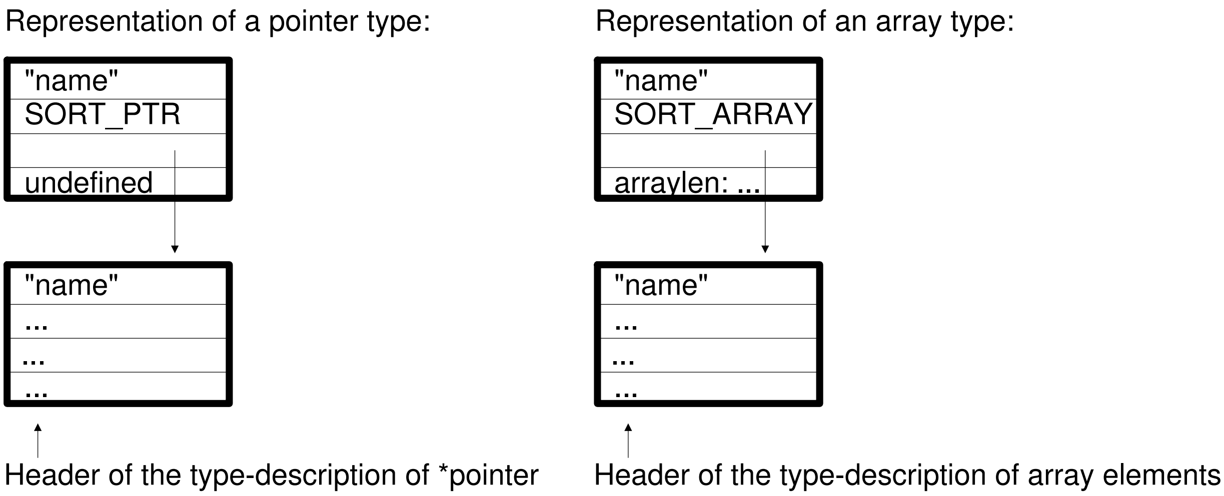


Fig. 2.5

A single type in the type table is described by the index of the type description header. For primitive types (int, char, float ...) this header provides a full type description (tsort = SORT_PRIM, tname = name of the primitive type). All other types require additional information about their subtypes. For pointers and arrays, the index of the subtype header is given in the *tptr* component is shown in fig. 2.6.



<u>Notes:</u>

The pointers shown here are actually indices in the type table
The size component is not shown in this drawing

Fig. 2.6

For struct and union types the *tsel* component is the index of the header for the first struct/union element (see fig. 2.7). The *tsel* field of this header is the index of the next struct/union element and so on. An entry INDEX_NULL in *tsel* marks the last element.

Representation of a structure (or union):

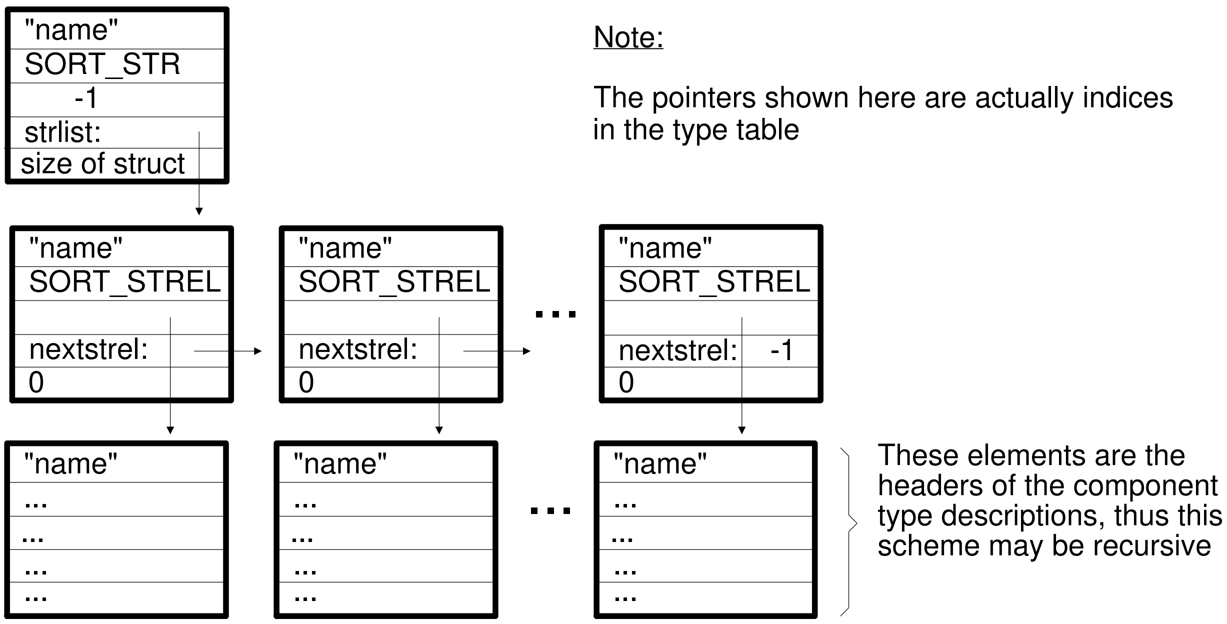


Fig. 2.7

An enum type is represented by a header of type SORT_ENUM whose *tsel* component is the index of the first enum element. The enum elements are chained via the *tsel* component similar to struct/union elements, but the *tsel* component also contains the numerical value of the enum element (see fig. 2.8)

Typel-array for an enum Type:

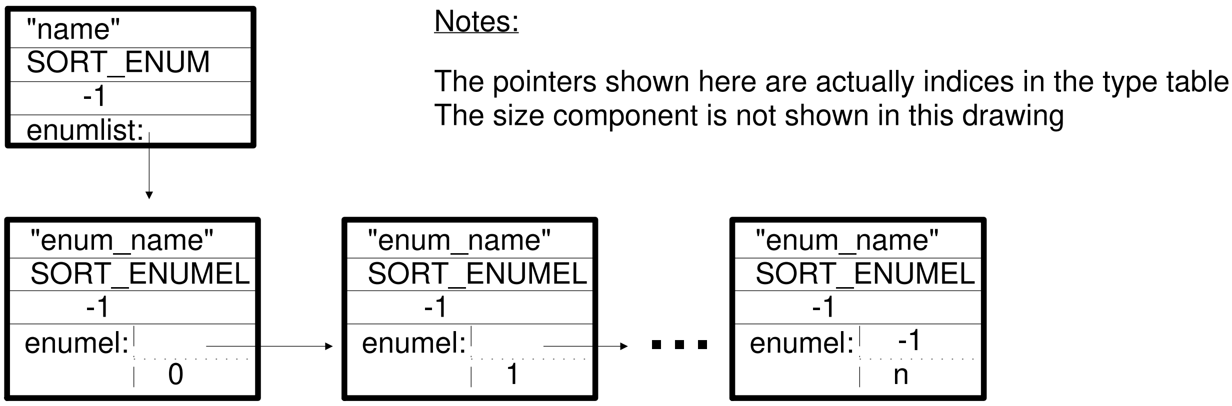


Fig. 2.8

Some more information about the type table is given in the appendix.

Example:

Assume that the application program possesses a module named "test" with the following source code:

```
static int i;
static struct str {
        enum color {red, blue = 2} col;
        char c;
} s;

static long fkt(pu)
unsigned pu;
{
        static char *p;
        struct str as[2];
        /* ... */
}
```

The HLTL symbol table for this program is shown in fig. 2.9.

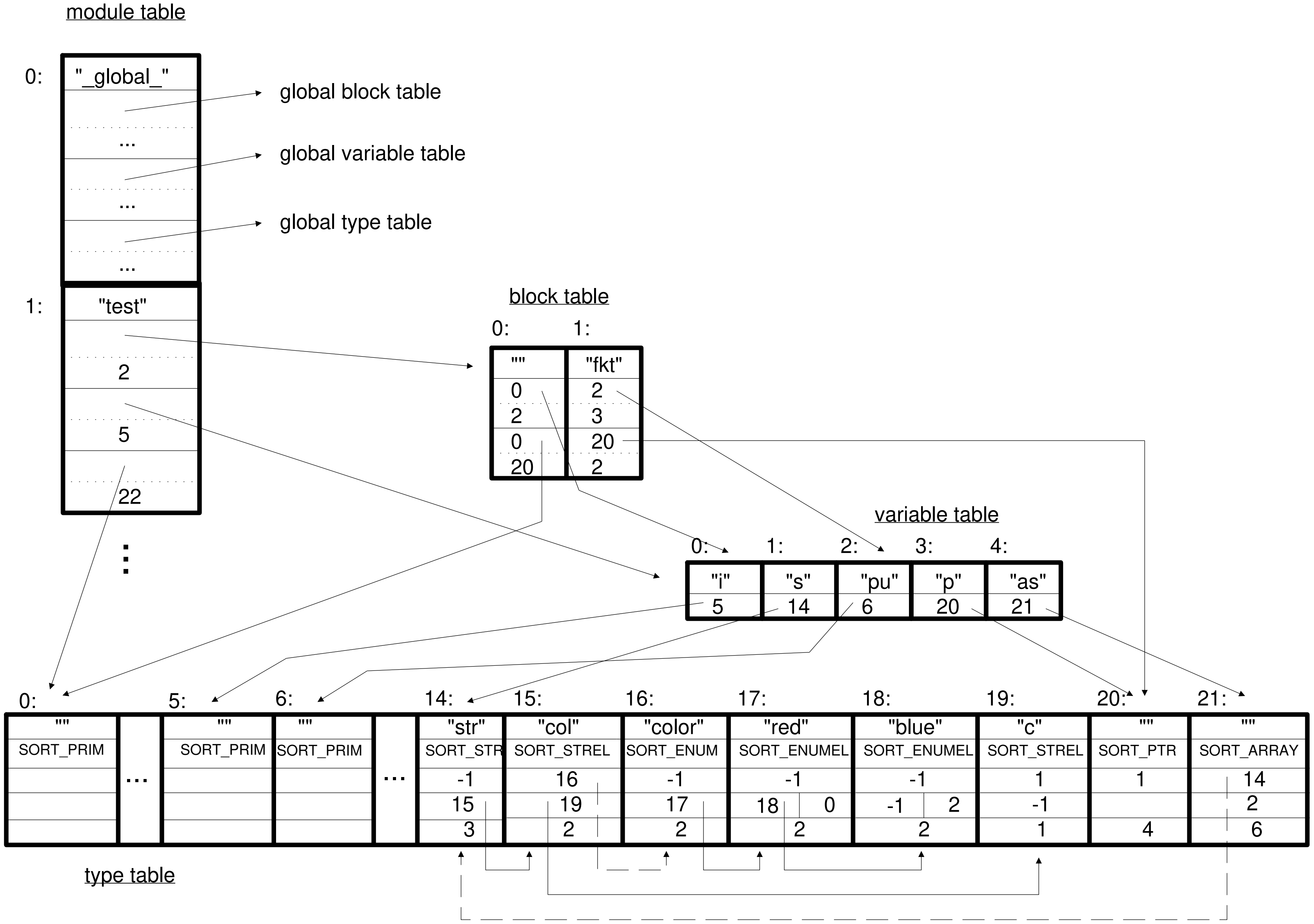


Fig. 2.9: Symbol table example

2.3.2 Initialization of the HLTL Symbol Table

To get symbol information from the compiler, the application program must be compiled with the '-g' option. The compiler then generates a symbol table in the Common Object File Format (COFF) and includes it in the application's executable file. Because the node compiler runs only on 80386 based workstations, but the tools should be available on any UNIX workstation, the UTOL cannot read the COFF file directly. Therefore we have built a converting program named 'read_symbols' that transforms the symbol information in the COFF file into an ASCII file that is transferred to the machine executing the tools. The program first reads the COFF file, using a special library that is only available on the compile machine, and stores the symbol information in an internal data structure. In the next pass this data structure is flattened and is written in a linear form to an ASCII file. This file has the name of the COFF file with the extension '.symb'. The format of the symbol file is shown in figure 2.10. It is much like the COFF format, but ASCII rather than binary coded, extended with a few entries containing the numbers of modules, blocks, symbols and types. These entries are used by the HLTL to allocate memory before building the symbol table.

When the HLTL reads the symbol file it builds the same data structure as used in 'read_symbols' to store the file contents. In a second pass the block, variable and type tables for every module are initialized using this data structure. The module, block and variable tables are built in a straight forward way, while generating the type table is more complex due to the recursive type definition scheme. Therefore the type table is built by a recursive procedure that calls itself each time a subtype has to be entered. A table of all already entered types is maintained, so that the procedure doesn't run into an infinite loop when entering a recursive data type.

Number of variables	
Number of types	
Number of blocks	
Number of modules	
	Blocklist (name of block = name of module)
	Pointer to next module
until pointer to next module == 0	

Blocklist:

	Name of block
	Pointer to list of symbols (if <>0, the list follows immediately, if ==0, list is empty)
	Symbollist
	Pointer to next block (see below)
	Number of types in this block
	Number of variables in this block
	Line number of the first statement in the block,
	Line number of the last statement in the block,
	Address of the first statement in the block,
	Address of the last statement in the block,
	Linelist
until pointer to next block == 0	

These components are zero if the Blocklist entry describes a module

Symbollist:

	Name of symbol (variable or type)	
	Address of variable or value of an enum type element	
	Section number (-1 = addr. is offset, 0 = addr. is absolute)	
	Type of the symbol	
	Storage class of the symbol	
	Number of auxiliary entries in the COFF file	
	Index of the type in the COFF file	
	Size of the type in bytes	
	Size of dimension 1 of an array	
	Size of dimension 2 of an array	
	Size of dimension 3 of an array	
	Size of dimension 4 of an array	
	Pointer to the subtypes or components of a complex type	
	if pointer <>0:	
		Symbollist (for subtypes, last symbol is named .eos)
	Pointer to the next symbol	
until pointer to the next symbol == 0		

If the index is nonzero and is used for the first time in the current module, the type specification follows.
In subsequent uses in the same module, the type is referred to only by this index.

Linelist:

	Address of first statement of the line
	Line number
	Pointer to next line
until pointer to next line == 0	

<u>Note:</u>
The pointers in the symbol file are only used as flags, i.e. they are only tested if zero or not.
The coding of the Symbollist components is the same as in COFF.

Fig. 2.10: Structure of the symbol file

2.3.3 Access Procedures

A couple of procedures which perform all data retrieval are attached to the HLTL symbol table, performing all data retrieval. The following functions are needed in the HLTL:

- Search a module by name
  Result is the module's index in the module table
- Search a block by name in a given block table
  Result is the block's index in the block table. This function is called before the C-parser is initialized to pass the index of the current block and the corresponding type table to the C-parser. In this way the active block (i.e. the current scope of variables) is determined. Searching for variables must be performed first in this block and then in the '_global_' block according to the C scoping rule.
- Search a variable by name in a given variable table
  Result is the index of the variable's type in the type table
- Determine the name of an enum element for a given integer value and a given enum type
  Result is the index of the element in the type table
- Determine the integer value of a given enum element in a given block
  Result is the integer value
- Search a struct/union component belonging to a given struct/union type
  Result is the component's index in the type table and the offset of the component in the structure.
- Search a struct/union component in all struct/union type definitions
  This functions allows for the 'illegal member use' that is possible in C.

The structure of the HLTL symbol table leads to a hierarchical search (module -> block -> variable -> type). In a given table a linear search algorithm is used because the number of table entries is relatively small and therefore it is not profitable to use expensive sorting to support faster search algorithms.

2.4 Handling of Complex Data Types

Complex data types (e.g. structures and arrays) are handled using the type table and its access functions. There are mainly two operations dealing with complex data types:

Component selection is performed by the C-parser. When it detects a selection, such as 'str.comp' it calls one of the symbol table search functions to search the component by name. The result is the offset of the component and a description of its name. This information is used by the parser to build a *Code_el* element for the component. References to array elements are handled directly by the parser which performs index calculation to get the element's offset.

The type and value descriptions returned by *type_s5()* and *display_s5()* are built by recursive functions that interpret the type table entries of the requested type. For non-primitive types a description of the type structure is built, mainly by copying the information from the type table, and recursive calls are performed to handle the subtypes. The recursion terminates for primitive types. If a variable value is determined for *display_s5()* the UDTL function *display()* is called to get the value of one component with a primitive type. The offset of this component is calculated and maintained during the interpretation process. This strategy has the advantage of great flexibility and makes the lower layers independent of the HLL used, but its major drawback is that every component of a data structure is retrieved individually through UDTL calls, making the *display_s5()* function relatively slow for large data structures.

2.5 An Example

This chapter describes the control and data flow in the HLTL with the help of a small example. Assume a call to the *display_s5()* procedure to get the value of the expression x[a.c1+3] in the context given by task tA, that is declared in module mA, and procedure pA in module mB, where x is an array of structures. The task and variable description passed to *display_s5()* are concatenated and analyzed by the display command parser. It uses the CHECK procedures, that in turn call the SEARCH procedures to search the module, task and procedure in the HLTL symbol table. Then it builds a *path_list* structure describing the context for evaluating the expression and passes both path list and expression to the C-parser. The C-parser analyzes the expression, calls the SEARCH functions to get information about the variables x and a and the offset of component c1 in structure a. Because the value of a.c1 is needed to evaluate the index, the C-parser uses the interpreter to get this value via the UDTL *display()* function. When the addition (+3) is performed and the element's offset in the array is determined the C-parser finishes. The parser's result, a *Code_el* structure describing how to access x[a.c1+3], is passed to *int_display()* that interprets the type information belonging to x. It calls *display()* for each component of x[a.c1+3] and builds a array of *Valel* structures that contains both the value of the expression and the value's structure. This array is then returned to the caller of *display_s5()*.

Figure 2.11 shows the data and control flow of this example in a more detailed graphical representation.

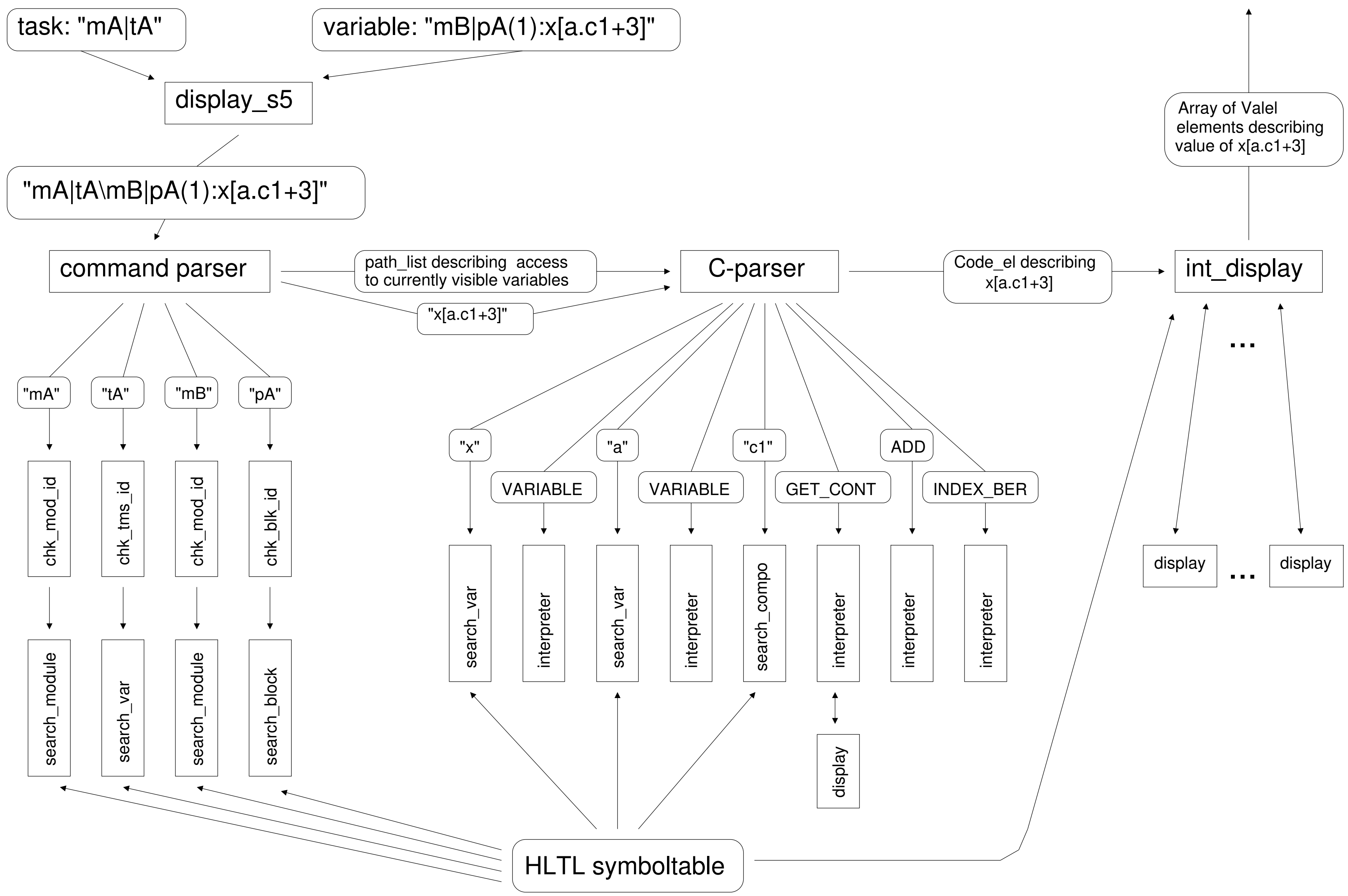


Fig. 2.11: Simplified flow diagram for a display command

3 The Universal Debug Tool Layer (UDTL)

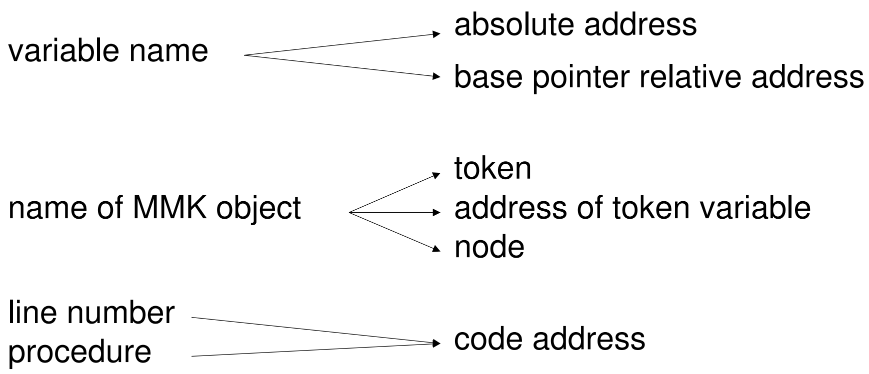
3.1 Overview

The UDTL (Universal Debugging Tool Layer) is mainly used for three tasks:

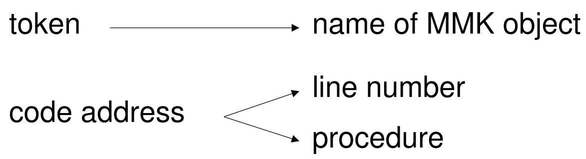
1) Transformation between the abstraction level of the tools and the machine level

The objects that are used in the S4 interface are names for modules, procedures, variables and MMK objects as well as line numbers in the source code. All parameters exchanged via S4 are composed of these objects. The monitoring system, however, works at the machine level. This means that it uses absolute and base pointer relative addresses to access data, absolute addresses for code and tokens for MMK objects. The number of the relevant node can also be regarded as an object of the monitor interface, although it does not appear in the interface definition. But the node number is implicitly used when the UDTL communicates with a specific monitor.

Therefore the UDTL has to perform the following transformations:



and the other way round:



2) Management of monitor resources

The UDTL is responsible for keeping track of all resources of the monitors that are currently used. In particular, the UDTL has to manage the definition, deletion, starting and stopping of traces and performance measurements. For this reason each trace and performance measurement gets a unique identifier at definition time. The UDTL must maintain a mapping between these identifiers and the monitor resources used. Because many tools may use the UTOL at the same time and different measurements may use the same resources, this is not a one-to-one mapping.

3) Communication with the target

The lower layer of the UDTL performs all communication with the distributed monitoring system on the target machine. There are three different communication schemes:

- Synchronous communication with a single node
  This one is used for local actions, such as 'inspect a task' or 'display a variable'.
- Synchronous communication with all nodes
  Communication with all nodes is needed if the application program has to be started or a breakpoint is defined. There are some severe problems with this form of communication that are explained in chapter 3.7.
- Asynchronous messages from the monitor
  A monitor sends an asynchronous message to the UDTL if it detects satisfaction of the breakpoint predicate or in case of some error conditions.

3.2 The Modules of the UDTL

The UDTL is divided into two sublayers. The lower of them is a communication layer that offers the monitoring system's functionality at the procedural interface S3. The upper layer consists of the S4 interface procedures, a module that performs evaluation and management of performance measurements and the transformation module maintaining the relationship between names on one hand and addresses and tokens on the other hand.

The monitoring system provides three different methods to acquire performance values: counters, integrators and performance traces. The MEASURE module decides which acquisition technique is used. It is supported by a module that keeps track of data acquisition done by the monitors and another module that manages the performance measurements defined via S4.

The transformation module consists of a submodule TRANS_UP that converts addresses and tokens into names and a module TRANS_PATH for the opposite direction. On top of it there is a module transforming breakpoint and trace predicates to the monitor level (i.e. machine level). These modules use the UDTL symbol table management (SRCH) and the LOCALIZER, mapping object names to processor node numbers.

The S3 interface procedures are built upon modules to perform binary to ASCII conversions (SUPPORT), localizing of tokens (LOCALIZE) and the communication with the monitors (SEND_CUBE).

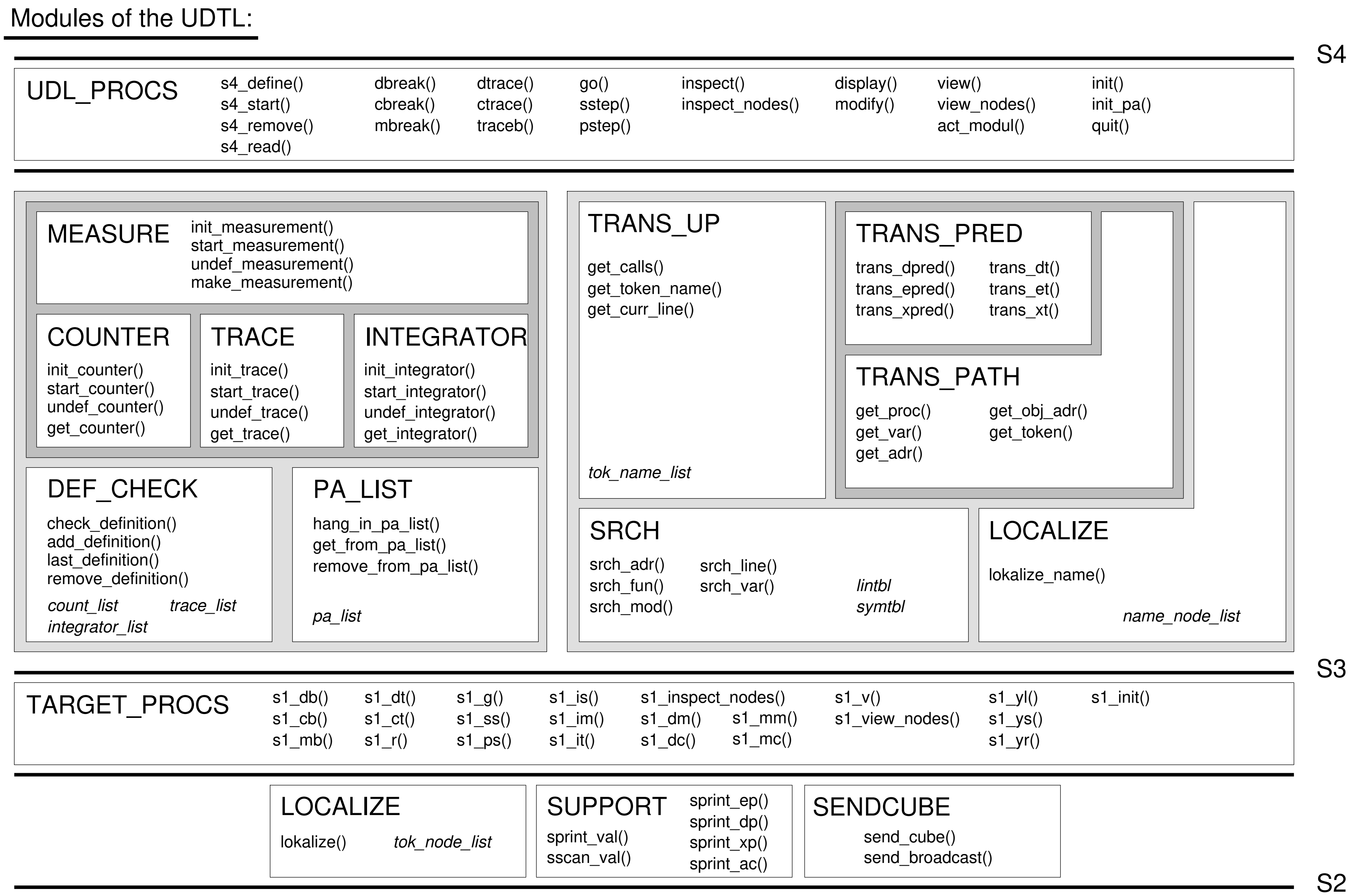


Fig. 3.3: Modules of the UDTL

3.3 Transformation between HLL Level and Machine Level

3.3.1 The UDTL Symbol Table

In order to perform the mapping between variable names and their corresponding data addresses and between line numbers or procedures and code addresses, the UDTL uses a symbol table that contains all the necessary information. The next three chapters describe the structure, initialization and the access functions of the symbol table.

3.3.1.1 Structure

The UDTL symbol table is a data structure consisting of several linked lists that form a tree structure with the global variable *symtbl* as a pointer to its root. The basic structure of this tree is shown in figure 3.4. The structure can handle nested function declarations (like in PASCAL), but in the current implementation that supports only C, this feature is unused.

To simplify the structure of the symbol table, modules and blocks (functions and procedures) are treated in nearly the same way. Therefore we need only three different structures to build the symbol table:

The *Fun_rec* structure describes blocks as well as modules and the special "_global_" module. The *bst* component is used to distinguish between them. The "_global_" module that represents the whole program has a *bst* of -2, the other modules get *bst*=-1. Blocks have a *bst*>=0 according to their nesting level. Each record contains a pointer to the next module/block, a pointer to the list of variables declared in this module/block and pointers to the static predecessor and successors. The *Fun_rec* records for blocks additionally contain the starting and end address of the block, a pointer to the block's module and pointers to the beginning and end of the line list.

The variable lists are implemented as linked lists of *Var_rec* structures. Each list element contains the name, the address and the storage class of one variable. The storage class determines whether the address is absolute or relative to the base pointer register. No information about the variable's type or its size is stored in the UDTL symbol table, because this information is handled completely by the HLTL.

The *Line_rec* structures, that build a linked list of lines for each block, contain the line number, the address of the first machine instruction of this line and a pointer to the block containing the line.

In order to speed up the search for a line number corresponding to a given code address, an auxiliary table, named *lintbl*, has been introduced. This array contains pointers to all *Line_rec* entries and is sorted by the code addresses of the lines. A similar array, named *funtbl*,

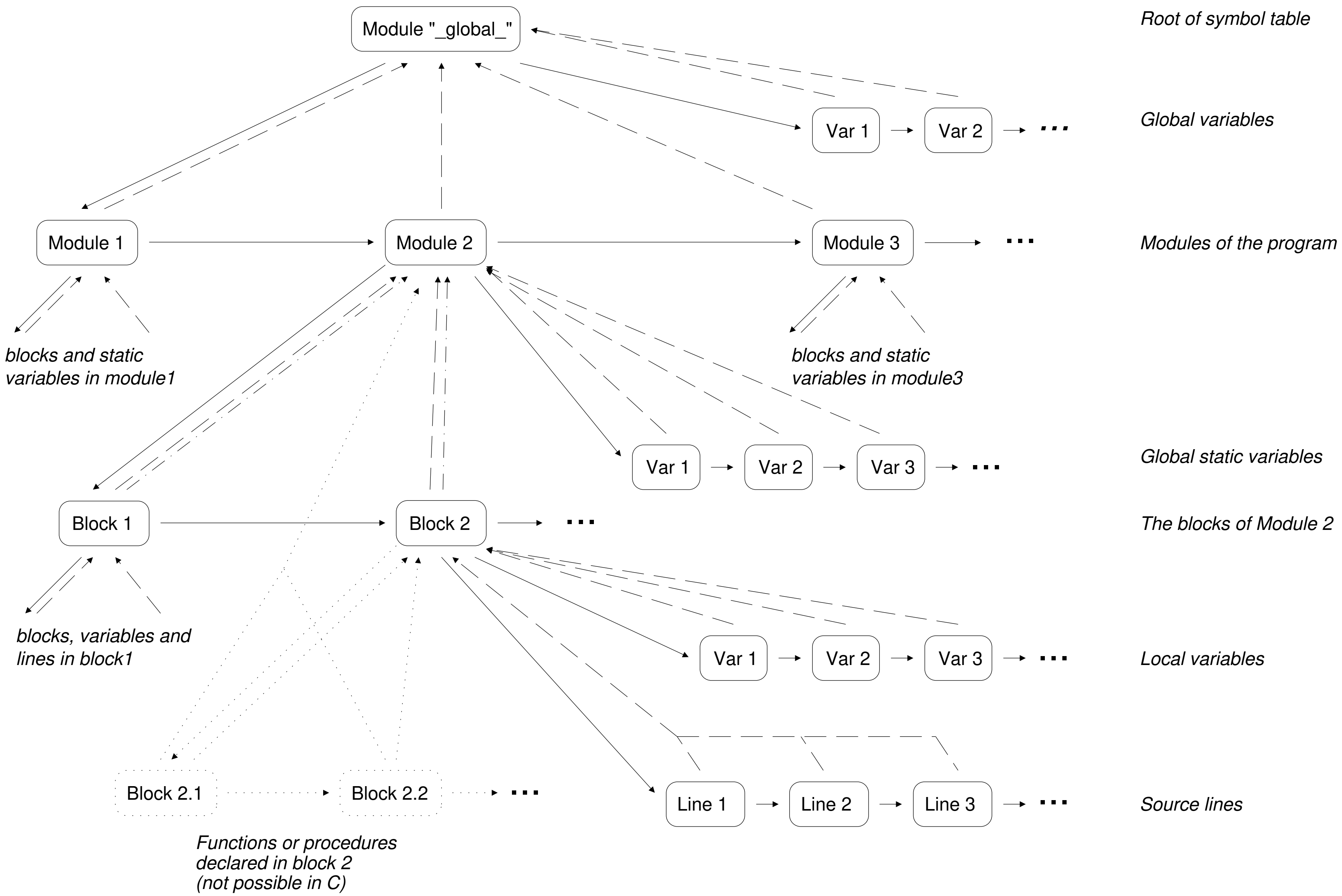


Fig. 3.4: Structure of the UDTL symbol table

holds pointers to all *Fun_rec* entries for blocks, sorted by the starting address of the block. The current implementation doesn't use this table, because it determines the block for a given code address by searching the corresponding *Line_rec* that contains a pointer to the block.

More details about the *Fun_rec*, *Var_rec* and *Line_rec* structures are shown in figure 3.5, an example for the symbol table is given in figure 3.6.

Structure of a Fun_rec:

name	*The name of the module or block*
next_fun	*Pointer to the Fun_rec of the next module or block*
s_line	*Pointer to the Line_rec of the first line of a block*
e_line	*Pointer to the Line_rec of the last line of a block*
s_pred	*Pointer to the Fun_rec of the static predecessor*
s_succ	*Pointer to the Fun_rec of the first static successor*
module	*Pointer to the Fun_rec of the module containing the block*
vars	*Pointer to the first Var_rec for the variables of the module or block*
bca	*Starting address of the block*
p_end	*Ending address of the block + 1*
bst	*nesting level ("_global_" -module = -2,, module = -1, block >= 0)*

Structure of a Var_rec:

name	*Name of the variable*
next_var	*Pointer to the Var_rec of the next variable in the same module or block*
proc	*Pointer to the Fun_rec for the module or block containing the variable*
adress	*Absolute address (global or static v.) or base pointer offset (local v.)*
class	*Storage class (local, global, static, ...)*
deflin	*not used*

Structure of a Line_rec:

lino	*Line number of this line*
next_line	*Pointer to Line_rec of next line*
s_ad	*Address of the first statement of this line*
proc	*Pointer to Fun_rec of the block containing this line*

Fig. 3.5: Structure of the UDTL symbol table records

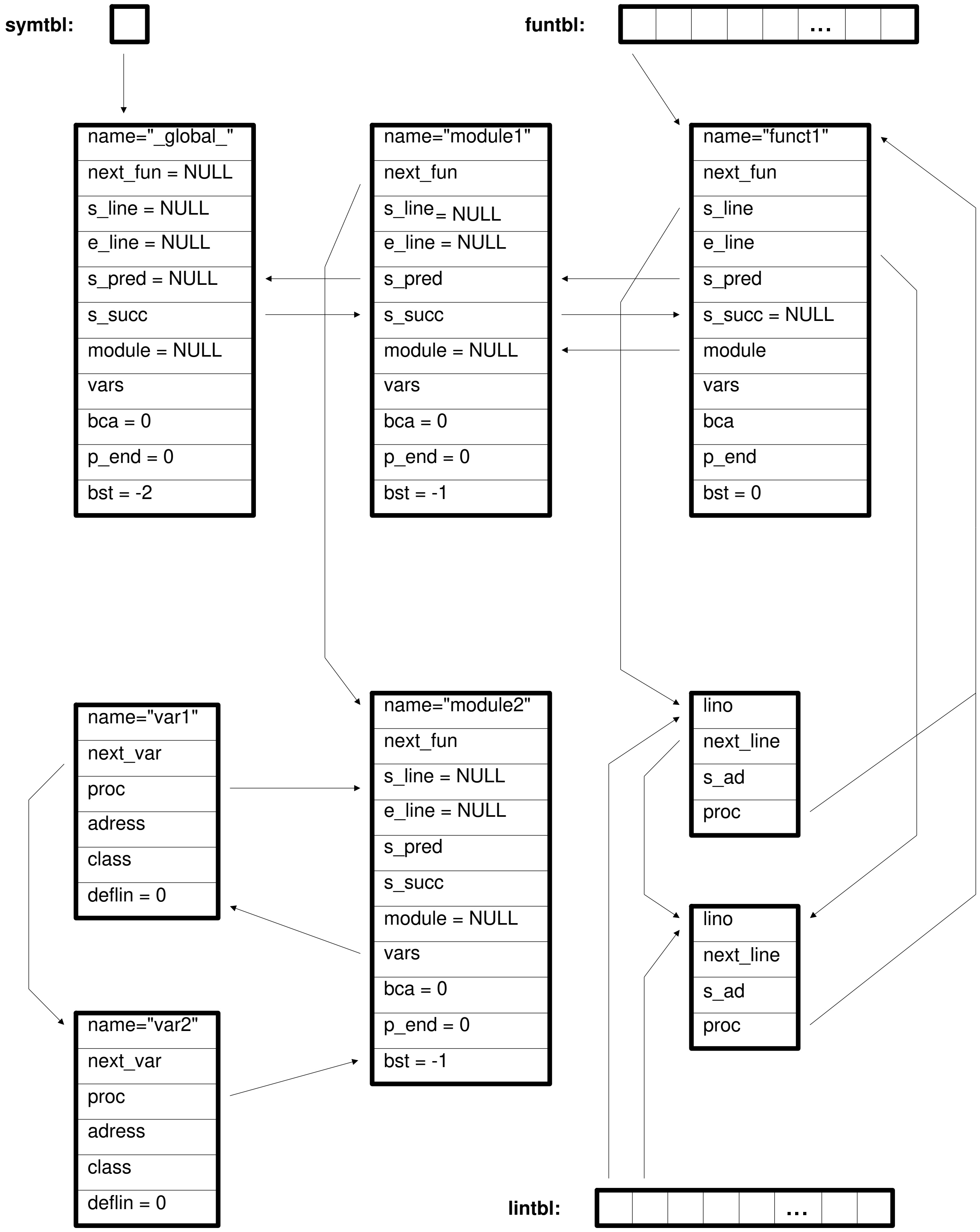


Fig. 3.6: Example of a UDTL symbol table

3.3.1.2 Initialization

The UDTL symbol table is initialized in the interface procedure *init()*. First the symbol file (see chapter 2.3.2 for a description of this file) is read into an intermediate data structure that is exactly the same as in the case of the HLTL. Then the root of the symbol table, a *Fun_rec* element for the module "_global_", is created. A list of *Var_rec* structures for all global variables is built and is hooked to the "_global_" *Fun_rec*.

Next, for each module the following steps are performed:

- Build a *Fun_rec* element for this module and append it to the static successor list of the "_global_" element.
- Build a list of *Fun_rec* structures for all blocks within the module. For each block a *Var_rec* list of local variables and a list of *Line_rec* structures is built.

When the actual symbol table is initialized, the two auxiliary tables are set up. The *lintbl* array is initialized by a recursive walk through the symbol table, starting at the root element "_global_", in the following order:

- enter lines of all static successors (i.e. all blocks declared within the current module/block)
- enter all lines of the current block by scanning the *Line_rec* list
- enter lines of the other blocks with the same nesting level

This scheme guarantees that *lintbl* is sorted by code addresses, but is of course in some way compiler dependent. The *funtbl* array is initialized in a very similar way.

3.3.1.3 Access Procedures

The SRCH module of the UDTL provides five functions to access the symbol table:

- *srch_mod()* searches a module by name and returns a pointer to the module's *Fun_rec*.
- When given a pointer to a module's *Fun_rec* and a block name, *srch_fun()* searches the *Fun_rec* for this block and returns a pointer to it.
- *srch_var()* is used to search a *Var_rec* for a given variable name. It needs a pointer to the *Fun_rec* structure of the module or block, in which the variable is declared, as an additional argument.
- *srch_line()* searches a *Line_rec* for a given line number within the module determined by a pointer to its *Fun_rec* entry.
- *srch_adr()* performs a binary search in the *lintbl* array to determine the *Line_rec* for the source line that corresponds to a given code address.

The transformations described in chapters 3.3.3 and 3.3.4 are based on these functions.

3.3.2 The Localizer

The task of the localizer is to determine the number of the processing node, a given MMK object (task, mailbox or semaphore) is mapped on. There are two versions of the localizer: One of them (*lokalize_name()*) determines the node number when the object's name is given, while the other one (*lokalize()*) needs an object token as input.

The current implementation is only suited for static objects. This means that the UTOL can only handle those objects that are declared in the object description file (*.global). In addition, these objects must not be deleted or migrated.

So the localizer mainly interprets the object description file. During the initialization phase of the UDTL, after the symbol table has been created, the object description file is read and the *name_node_list* is built. For each static object there is an element that contains the object's name, its type (task, mailbox or semaphore) and the number of the node it is mapped on. At the same time a list of all used nodes (*node_list*) is created. This list is needed by some commands (e.g. "Run All") and for the initialization of the monitors.

The mapping between tokens and nodes is determined by asking the monitors on each used node to return all tokens on that node. This is done only once during the initialization. The resulting pairs (token,node) are stored in another list named *tok_node_list*. In principle the *tok_node_list* could also be derived from the *name_node_list*, but the strategy used is easier to extend to suspect dynamic objects.

3.3.3 Transformation from HLL Level (S4) to Machine Level (S1)

This transformation is performed by the two modules TRANS_PRED and TRANS_PATH. The latter converts variable names into absolute data addresses and object names into tokens, TRANS_PRED uses these functions to transform breakpoint and trace predicates.

The S4 interface uses so-called "path lists" to describe the access paths to variables. A path list is a linked list, where each list element has the following structure:

obj_type	*Type of the path list element*
obj_name	*Name of the path list element*
off_ink	*Offset for variables, incarnation for blocks*
next	*Pointer to the next element*

Possible types are:

T_module	*Module*	*off_ink = don't care*
T_funcpro	*Block*	*off_ink = incarnation of the procedure or function*
T_task	*Task*	*off_ink = 0*
T_var	*Variable*	*off_ink = offset from the variable's base address*
T_access	*Access*	*off_ink = offset from the new address*

Fig. 3.7: Path_list structure

A path list for a global variable has the following form (in the rest of this chapter path lists are represented as sequences of strings):

*module task module variable {access}**

The path list "mod1 task1 mod2 glob1" for example is a representation for the global variable glob1, declared in module mod2, in the address space of the task task1 declared in module mod1.

A path list for a local variable is more complex. To reference a local variable the complete call sequence to the block that contains the variable is needed. Because the UDTL can handle nested blocks, the following general form arises:

module task $\{module\ \{block\}^{+}\ \}^{+}$ *variable {access}**

The sublists "module $\{\text{block}\}^{+}$" reflect the static path to a block. For example, "module1 block1 block2" stands for block2, that is declared in block1, where block1 is declared in module1. Of course, this kind of nesting procedure or function declarations is not possible in C. For a block in C the static path always has the form "module block".

The sublist "{module {block}$^{+}$ }$^{+}$" is the complete dynamic call sequence leading to the requested block, starting with the main routine.

A special format exists for variables that are visible in the current scope of a procedure, where the exact call sequence is not important:

module task module {block}$^{+}$ *variable {access}**

This path list describes the variable with specified name in the last incarnation of the block, where the variable can be declared in the block itself or in any of its static predecessors (outer blocks). To distinguish this format from the general form, the last block in {block}$^{+}$ must have the *off_ink* component set to 0.

The access elements are used to represent pointer dereference. So the C expression "*(i+9)" is represented by a path list ending like this:

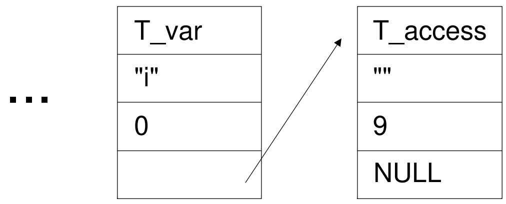


Fig. 3.8

The TRANS_PATH module contains a function (*get_adr()*) that determines the absolute address for the variable given by a path list. For local variables the absolute address depends on the current base pointer chain and therefore on the current call sequence. So it is necessary for *get_adr()* to access the monitor on the variable's node to determine the address.The node number is ascertained from the task specification in the path list using the *lokalize_name()* function.

Then for each element ("module {block}$^{+}$") of the dynamic call sequence the block's *Fun_rec* entry in the symbol table is determined using *srch_mod()* and *srch_fun()*. The BCA (Block Code Address = starting address of the block's code) component is extracted from the entry and is appended to the BCA list.

Then the variable is searched in the variable list of the last block (*srch_var()*). If it has not been found and the path list specifies a variable that is visible in that block (indicated by *off_ink*=0), the variable is searched in the block's static predecessors also.

If the symbol table entry for the variable has been found the address of a global or static variable can by extracted from the entry's address component. For local variables this component contains an offset from the base pointer. Because only the monitor can determine the appropriate base pointer value, the BCA list and the offset are passed to the monitor with an YL command. The monitor then compares the BCA list with the current call sequence. If these sequences match exactly the monitor returns the current absolute address of the variable. For a local variable in the current scope the monitor's YV command is used and only the BCA of the last block is passed to the monitor instead of the whole list. See part 5 of the design documents for a more detailed description of these monitor commands.

When the address is determined, the access elements are evaluated to get the final address. Each access element results in a call of the monitor's DM command for the necessary memory access.

To convert an object name into the object's token, a function named *get_token()* is provided. This function is based on the convention that the name of an object is the name of a global or global static variable that contains the object's token. The MMK preprocessors generate these variables for each object in the object description file. So the transformation is very easy: search the symbol table for a global or global static variable with the given name, use *lokalize_name()* to determine the appropriate node and call the monitor's DM command. The result of this command is the object's token.

The module TRANS_PRED contains functions that translate breakpoint and trace predicates from S4 level to S1 level. They work by transforming each S4 object (block, variable, line number, MMK object name) to the corresponding S1 object (BCA, data address, code address, address of token variable). The S1 predicates contain no token but the address, where the token is stored. This facilitates the implementation for handling dynamically created MMK objects, because the object (and its token) need not exist when the breakpoint or trace is defined.

3.3.4 Transformation from Machine Level (S1) to HLL Level (S4)

There are three transformations that have to be done by the TRANS_UP module: It must convert object tokens into object names, code addresses into line numbers and BCA's into block names.

As stated in the last chapter, an object name is the name of a variable that contains the object's token. If we would use this convention for transforming tokens into names, we would have to to look at each global or global static variable of the appropriate in order to check type if its contents equals to the token's value. This would be extremely inefficient and could lead to a wrong name if a variable contains the token's value by chance. For these reasons we have chosen a different method:

During the initialization of the UDLT, after the *name_node_list* (see 3.3.2.) has been created, a list containing the names and the associated tokens of all static MMK objects is built (*tok_name_list*). This is done by traversing the *name_node_list*. For each name in the list, the address of the corresponding token is determined as explained in chapter 3.3.2. The content of this address is requested from the monitor with the DM command. The appropriate node is taken from the *name_node_list*.

The *get_token_name()* function then simply searches the *tok_name_list* for the given token. If the token is in the list, the name in the list is returned, if not, a name like <token|7f9> is generated (if 7f9 is the hexadecimal value of the token).

The line number, that contains a given code address is determined using *srch_adr()*. The *Line_rec* structure returned by this procedure contains the line number. The block belonging to a BCA is determined by searching the line that contains the BCA (this is the first line of the block). The line entry in the symbol table contains a pointer to the block entry, where the block's name is stored.

To ascertain the block that is currently executed by a task, the current call sequence is requested from the monitor with the YS command. The result of this command is a list of the BCAs of all blocks in the call sequence. In addition to the user visible blocks (these blocks have a symbol table entry) this list contains the BCAs of library functions. To build the call sequence at user level, for each BCA in the list the corresponding block is searched. If a block is found, its name is added to the retranslated call sequence. So this sequence contains only user visible blocks.

The current line of a task is determined by passing the BCA of the last visible block to the monitor with the IT command. The monitor then returns the current address in this block. This address is then translated into a line number.

3.4 Breakpoint Processing

3.4.1 Introduction

In the TOPSYS environment a breakpoint definition always consists of a breakpoint predicate and of actions that have to be performed when the predicate holds. The breakpoint predicate is composed of simple predicates with conjunctive (+) and/or sequential operators (->). The sequence operator has precedence over the or operator. The semantics of the operators are defined as follows:

a + b holds	<=>	a holds or b holds
a -> b holds	<=>	a holds at time t0 and b holds at time t1 > t0 + dt, where dt is implementation dependent. In other words: the events corresponding to a and b must occur in the order first a and then b.

The UTOL offers three kinds of simple predicates:

- execution predicates describe control flow events like “task A begins execution of line 15” or ”task B enters procedure foo”.
- data predicates are triggered by accesses to variables. Access direction can be either read or write.
- exchange predicates concern the message transfer to and from mailboxes and semaphore accesses.

A more detailed discussion of these predicates can be found in the DETOP User’s Guide.

The breakpoint predicate is passed to the UDTL as a linked list. Each list element contains a description of the simple predicate and the operator that follows this predicate. One task that the UDTL has to perform is the distribution of this predicate over the monitors. Each monitor can only supervise events on its own node and a set of special signals called crosstriggers (see the monitor documentation). They are used for communication between monitors on different nodes. The monitors must be properly programmed by the UDTL in order to generate and recognize the correct crosstriggers.

In addition to the distribution of the breakpoint predicate, the UDTL must also change the structure of the predicate. As stated above, the breakpoint predicate is defined as a disjunction of sequences, i.e. has the following general form (the parentheses have only been included to emphasize the operator precedence):

$(A_0\text{->...->}A_n) + (B_0\text{->...->}B_n) + ... + (Z_0\text{->...->}Z_n)$

The monitors, however, expect the predicate in a form that can be viewed as a state machine: It consists of so-called OR-blocks. Each OR-block contains several simple predicates that are supervised concurrently. Associated with each predicate is an action that is performed when the predicate is recognized as holding. In addition to the actions available at the UTOL interface (stop tasks, stop trace), it is possible to generate a crosstrigger and/or to switch supervision to another OR-block.

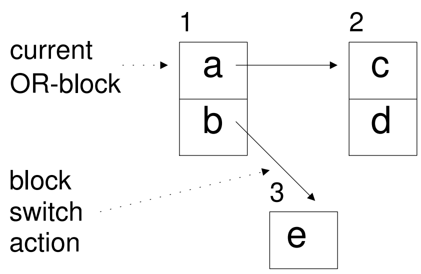


Fig. 3.9: OR-block structure of the monitor

Figure 3.9 shows an example of this structure. The monitor initially supervises the simple predicates a and b in block 1. If a is satisfied, it switches to block 2 and supervises the predicates c and d. Analogously, if b holds, the monitor switches to block 3.

3.4.2 Transformation Algorithm

The transformation of an S4 breakpoint predicate to the S1 breakpoint definitions for each monitor is performed in seven steps:

1) The first thing that is done is the conversion of all names in the predicate definition into addresses as explained in chapter 3.3.3.
2) For each simple predicate the number of the node whose monitor can supervise this predicate is determined. This is normally the node of the task that is involved in the predicate.
3) The breakpoint predicate is augmented with crosstrigger actions (generation) and crosstrigger predicates (recognition). The crosstrigger with number 0 (CT0) is always used to give notice of the fulfillment of the breakpoint predicate. This signal can be caused by any monitor. It therefore implements the or operation between predicates on different nodes. In order to detect sequences of events on different nodes, further crosstriggers are needed. Each subpredicate of the form a->b, where a and b are monitored on different nodes (say nodes 0 and 1), is implemented in the following way:

   monitor 0 supervises a; if a holds, monitor 0 generates a crosstrigger i;

   monitor 1 supervises the subpredicate CTi->b, where CTi is true, when crosstrigger i arrives ('monitor x' is a short form of 'the monitor on node x').

   After a crosstrigger is used in that way, i (starting at 1) is incremented. So each crosstrigger is used at most once, a fact that is not optimal if the hardware monitor is used, which has only eight crosstriggers.
4) In this step the augmented breakpoint predicate is broken into the local subpredicates for each monitor. The result is an array of lists for each node that represents the sub-

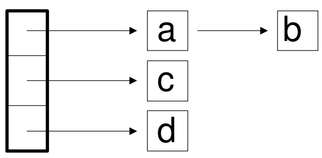


Fig. 3.10

predicate to be supervised on this node. The subpredicate a->b+c+d, for example, is represented by this array:

i.e. the lists represent sequences, while the array represents the OR operation.

This structure is built by scanning the resulting list of step 3. Each consecutive part of a predicate sequence, that consists only of predicates on the considered node, forms one entry in the array. The next example should clarify this:

Assume that the following list exists after step 3:

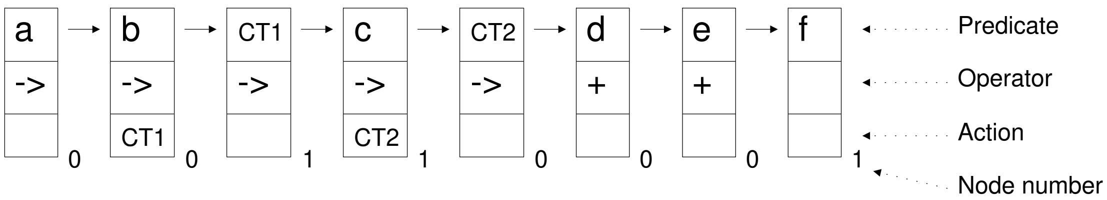


Fig. 3.11

This list is converted into the following arrays:

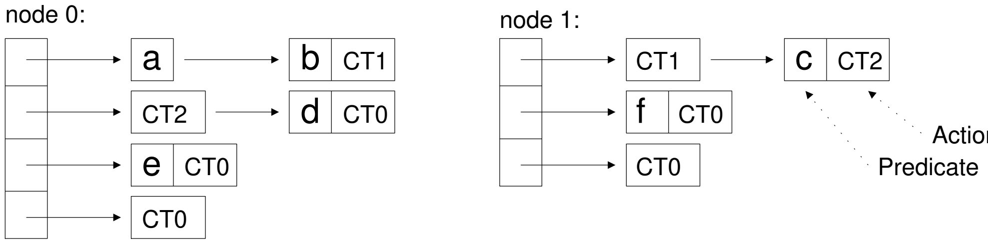


Fig. 3.12

The additional entry (predicate = CT0, action = 'stop tasks') is created to realize the or operator: if any of the monitors detects that the breakpoint predicate is fulfilled, it informs the other monitors with the CT0 signal.

5) The most complicated step is the transformation of the local subpredicates into the OR-blocks needed by the monitors. This step is performed for every node. The OR-block structure is represented as a tree in this step, but for the reason of convenience the example that follows shows not this tree, but the corresponding OR-blocks.

The tree is built from the array by the following recursive scheme:

- generate an OR-block that contains the first predicate of each sequence list in the array.
- for each simple predicate p in the OR-block:
  generate the tree that corresponds to the array when p is removed from it (recursive call). If this tree is not empty, its root is the destination of the block switch action associated with p. In other words, assumed that p has been fulfilled, determine the OR-block structure of predicates that must be supervised now.

Example:

Assume the predicate a->b+c->d, where all simple predicates are on the same node. Step 4 creates the following array:

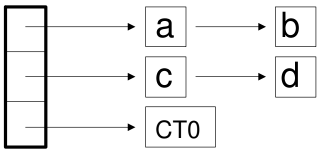


Fig. 3.13

The OR-block tree that results from step 5 looks like this:

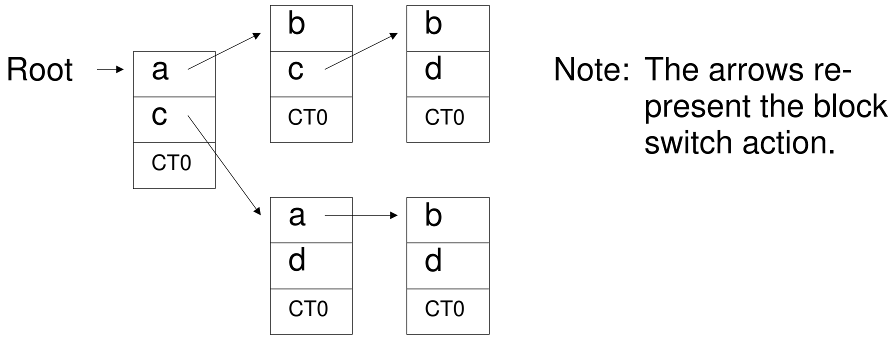


Fig. 3.14

It can be seen in this example that the algorithm isn't optimal, because one OR-block occurs twice. Further the algorithm does not handle the nondeterminism that arises if, for instance, a happens to be the same simple predicate as c.

Once the tree has been created it is converted into the linear form needed by the monitors.

6) After the predicate is completely transformed, the actions are converted. Currently only the 'stop tasks' action works. The only thing that must be done here is to convert the task names into addresses of tokens.

7) The results of 5) and 6) are sent to the monitoring system.

3.4.3 Breakpoint Notification

As stated before, the monitor that first detects satisfaction of the breakpoint predicate sends a CT0 signal to all other monitors. In addition it sends a breakpoint message to the host (with the NX *csend()* routine). During initialization of the UDTL the Unix process "a_sendcube" has been created, that waits for this kind of messages in an endless loop. If a message arrives, a_sendcube sends the Unix signal SIGUSR1 to the tool process. A handler function for this signal has been set in the HLTL *s5_dbreak()* function when the breakpoint was defined. This function is now called by the Unix signal mechanism. It should be mentioned that the function should not assume anything about the state of the tool process. In particular, the handler should not use any X window calls.

3.5 Trace Management

For the management of all the traces defined by a tool, the UTOL maintains a linear list of all defined traces. Each element in the list contains:

- The definition of the trace as passed to the HLTL. This definition is an ASCII string that is needed for the ’list trace’ function.
- The trace definition in a parsed form. It is used when the trace buffer is translated back to the HLL-level
- The number of the trace. This is the trace number that was passed to the tool when the trace has been defined.
- The number of the node whose monitor collects the trace data.
- The local trace number. This is the trace number that the monitor on the respective node has assigned to the trace.
- The trace buffer. It contains the trace data after the UDTL has read the trace buffer from the monitoring system. It is implemented as a circular buffer with an index to the first free entry (write index) and to the first valid entry (read index).

When a tool defines a trace, the UTOL generates a unique number for this trace definition. Afterwards it appends a new element with an empty trace buffer to the trace list. The UTOL does not check for a double trace definition (in such a case the data is simply recorded twice). Then the trace definition is translated to the monitor level (see 3.3.3) and is sent to the monitor on the appropriate node. The local trace number returned by the monitor is stored in the trace list entry.

When a trace is cleared, its entry is removed from the trace list. The node number and the local trace number in the entry are used to send a ’clear trace’ command to the responsible monitor. The trace buffer of the cleared trace is also freed, and is therefore no longer available.

If a tool requests the contents of the trace buffer for a given trace, the UDTL first searches the entry for this trace in the trace list. Then all other traces, that are defined on the same node, are determined. This is necessary because the monitor has only one buffer for all traces. So it is only possible to read the data of all traces on one node together (Note that the monitor’s trace buffer is cleared after reading; see part 5 of the design documents). After the buffer has been read from the monitor, each element in the buffer is processed. First the trace, to which the entry belongs, is determined. Next, the parsed definition of that trace is inspected to determine if the entry contains tokens or code addresses. If so, they are translated into names and line numbers. Then the entry is appended to the buffer of that trace.

For the reason of efficiency, the UTOL does not return a copy of the trace buffer to the caller, but returns a pointer to the trace's list entry. The tool must extract the trace buffer from this entry and must also modify the read and write indices if it wants to delete part of the buffer after reading it.

3.6 Performance Measurements

The UDTL converts the performance measurements requested by a tool at the HLTL interface into measurements that can be performed by the monitor system. To avoid confusion concerning the different types of measurements, in the rest of this document the measurements offered at the HLTL interface are called HL-(HighLevel)-measurements, while the measurements of the monitoring system are called LL-(LowLevel)-measurements.

For every requested HL-measurement, the UDTL decides on the data acquisition method (i.e. which LL-measurement type is to be used) and performs the evaluation of the measuring (i.e. transforms the LL-measurement results into HL-measurement results). It is possible for many HL-measurements to share a single LL-measurement.

To manage this task, the UDTL uses two modules (PA_LIST, DEF_CHECK) that maintain several lists. The *pa_list* is a list of all defined HL-measurements. Each element contains the following information:

- The *Tool_token* of the measurement (see the HLTL interface description)
- The definition of the measurement as passed to the HLTL
- A list of measurement results (these results are passed to a tool when it calls *pa_read()*)
- Evaluation information (see below)

For each of the three acquisition methods provided by the monitoring system (counters, integrators and traces), a list of all LL-measurements using this method is maintained. An entry in this list consists of:

- The monitor command to define this measurement
- The ID of the measurement (this ID has been returned by the monitoring system when the measurement was defined)
- Evaluation information

The evaluation information contains all data that is needed to transform the measurement results:

- The measurement method (counter, integrator, trace)
- The node number and the ID of the LL-measurement
- The host time, when the last LL-result has been received
- A list of all HL-measurements that use this LL-measurement

When a HL-measurement is defined, the UDTL first determines the acquisition method that is to be used (see table 3.1). Then it searches the counter, integrator or trace list, to find out whether there is already a LL-measurement defined with the same type and parameters. If so, the HL-measurement is added to the list in the evaluation information of this LL-measurement. Otherwise a new LL-measurement is defined; the definition is sent to the monitoring system and is inserted into the appropriate list. For time measurements using traces there are always two traces defined: one for the beginning of the operation in concern, the other for the end of the operation.

pa_type	Measurement Method
READYQUEUE IDLE_TIME QUEUE_ON_MBOX QUEUE_ON_SEMA QUEUE_ON_RPC	Integrator
SEND_BYTES	Counter
SNDMESSAGE RECMESSAGE	One trace recording message lengths
TASKRUNNING WAITSEND T_WAITREC M_WAITREC	Two traces recording begin and end of the operation in interest

Table 3.1: Performance measurement methods

To start a HL-measurement the corresponding LL-measurement is determined using the *pa_list*. If it is not yet started, a start command is sent to the appropriate monitor.

If a HL-measurement is deleted, the UDTL checks whether the corresponding LL-measurement is needed by other HL-measurements. If not, it is completely deleted. Otherwise the HL-measurement is only removed from the evaluation information of the LL-measurement.

When a tool requests new performance results, the UDTL looks if there are already new results due to the evaluation of another HL-measurement. If this is the case, these results are returned. In the other case the UDTL requests LL-measurement results from the monitor system, evaluates them and stores them in the result list of each HL-measurement that uses this LL-measurement. For counters and integrators the evaluation is a simple rescaling that produces exactly one performance value. The evaluation of the trace based measurements is more complex. In this case a list of performance values is generated. Each value concerns a time interval of a length given by the HL-measurement's granularity parameter.

As already stated, two traces are generated for time related measurements. Each trace entry contains a time stamp, the entries in the SNDMESSAGE and RECMESSAGE traces also contain the length of the message sent or received. The trace buffer received from a monitor contains the entries of all traces defined on the monitor's node in time ascending order. To distinguish between the different traces, the ID of the corresponding trace is affixed to each entry. So the UDTL cannot evaluate a single trace based measurings, but must process all traces on a node together. For each measurement the following operations are performed:

The trace buffer is scanned for the next entry that belongs to the currently considered LL-measurement. If the entry marks an "end event" (e.g. task stops sending), the time difference to the preceding "begin event" (e.g. task begins to send) is added to the current measurement value. For the SNDMESSAGE and RECMESSAGE traces the message lengths are simply summed up. When the current measurement value comprises a time interval of at least the length given by the granularity parameter, the value is appended to the HL-measurement's value list and a new value is assembled.

Example:

Consider the HL-measurement for the time task A spends sending messages. The corresponding LL-measurement consists of two traces. Trace 1 contains the points of time, when task A

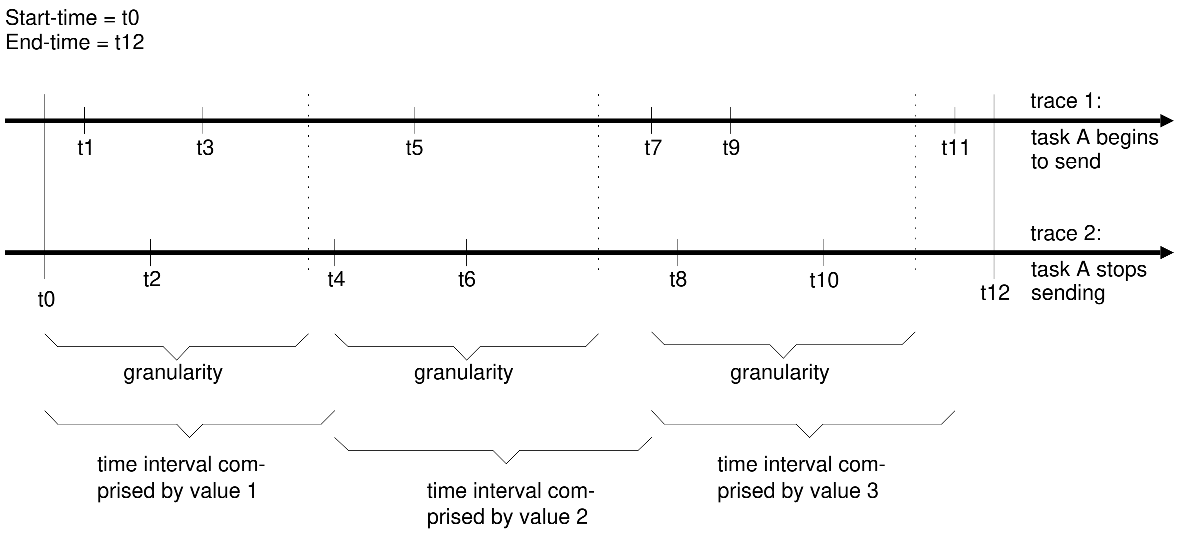


Fig. 3.15: Example of a trace-based measurement

begins to send, trace 2 records when task A finished sending. Fig. 3.15 shows what the two traces could look like when viewed as timing diagram.

So the first measurement value generated is:

(t2-t1)+(t4-t3)	Start time: t0	End time: t4

the next is

(t6-t5)	Start time: t4	End time: t7

and the last

(t8-t7)+(t10-t9)	Start time: t7	End time: t11

Note that the granularity parameter only specifies the minimum time interval comprised by one performance value.

3.7 Communication with the monitoring system

The communication between the UDTL and the distributed monitoring system is based on INTEL's Remote Hosting software. The UDTL sends commands to a monitor and waits for the monitor's response. All commands and results are transferred as ASCII strings (S2 interface). A couple of procedures (TARGET_PROCS) is provided to transform this command interface into a procedural interface. These procedures use the routines from the SUPPORT module to perform the binary to ASCII and ASCII to binary translations.

Most of the procedures in TARGET_PROCS only communicate with a single monitor. Here the communication scheme is a straightforward synchronous communication. The command string is sent with the *csend()* function, afterwards *crevc()* is called to receive the result from the monitor.

For some commands, however, the UDTL must communicate with more than one monitor. These are the 'run tasks', 'set breakpoint', 'clear breakpoint' and 'masterbreak' commands. Because each monitor gets different parameters (for instance different task tokens for the 'run' command), the command cannot be sent with a broadcast. So the command is sent to all monitors one after another. But with this scheme a problem can arise: Assume the definition of a breakpoint with predicate a->b, where a is located on node 0, b on node 1. Further assume that the breakpoint command is first sent to node 0 and afterwards to node 1. Then it is possible that predicate a is fulfilled soon after the breakpoint definition, so that the crosstrigger sent from node 0 arrives at node 1 earlier than the breakpoint definition. This means that the crosstrigger is lost and the monitoring system is in an inconsistent state. Similar problems can arise with the other commands mentioned above.

To avoid this problem, the monitors wait for a start message from the UDTL after one of these commands is defined. The start message is the same for every monitor, so it can be broadcast . Due to the implementation of the monitors no other task can be running while the monitor waits for the start message, so no events can occur during this time. But there are still some other potential problems with the commands concerning more than one monitor that are not solved until now.

The asynchronous communication scheme used for the breakpoint notification has already been explained in chapter 3.4.3.

Appendix: Description of the HLTL - Interface

Preliminary note:

All procedures of the HLTL - interface have a return parameter named *except*. This parameter is a pointer to a variable of type *Errmsg*, that contains either OK, when the procedure returned without an error, or an error message (see HLTL.h for a list of all possible error values). If the procedure returned an error, all other return values should be regarded as undefined. In particular, no *free()* operation must be performed on any pointer returned.

1 Breakpoint and Trace Commands

NAME:

dbreak_s5() - define a breakpoint

SYNOPSIS:

void dbreak_s5(except,predicates,stoptasks,handler)

Errmsg *except;	Pointer to return value
char *predicates[];	The concatenation of the strings in this string array defines the breakpoint predicate and must meet the syntax described below. The string array must be terminated with a NULL pointer.
char *stoptasks[];	String array defining the actions to be taken, when the predicate is satisfied. The syntax of these strings is given below. The string array must be terminated with a NULL pointer.
int (*handler)();	Pointer to the handler function that is called when satisfaction of the predicate is detected. No parameters are passed to this function.

STRING SYNTAX:

predicates:
predop

predop:
predicate
| predicate op predop

op:

'>'	sequential combination
| 'v'	OR
| 'V'	OR

predicate:	simple predicates
datap	data predicate
| execp	execution predicate
| exchp	exchange predicate
datap:	data predicate

task '\' locvar varcond
| task '\' globvar varcond

locvar:	
var	variable within current scope
| proc ':' var	local variable in current module
| module proc ':' var	local variable
globvar:	
module var	global variable
proc:	
procname	default for incarnation: 1
| procname '(' unsigned ')'	define procedure incarnation
var:	the resulting type of the variable defined here must not exceed 32 bit in size
varname	
| var indirection	
indirection:	
'[' index ']'	
| '.' strunkomp	
index:	
IDENTIFIER	member of an enum type
| unsigned	
varcond:	
'==' value	write access with specified value
| 'read'	read access
| 'write'	write access with any value
value:	
CHAR	
| LONG	
| FLOAT	
| IDENTIFIER	member of an enum type
execp:	execution predicate
linepred optcount	breakpoint at specified line
| procpred optcount	breakpoint at beginning of procedure
linepred:	
task '\' unsigned	line in current module
| task '\' module unsigned	line in specified module
procpred:	
task '\' procname	procedure in current module
| task '\' module procname	procedure in specified module
optcount:	
<empty>	
| '(' unsigned ')'	event counter

exchp: exchange predicate

taskspec sndrcv mailspec np
| taskspec reqrel semaspec units

taskspec:
mailspec:
semaspec:

<empty>
| '*'
<empty> and '*' mean don't care. If taskspec is don't care, mailspec or semaspec must not be don't care
| spec

np:

<empty>
<empty> and '*' mean don't care
| '*'
| var
variable in current module
| module var
variable containing the message pointer

units:

<empty>
<empty> and '*' mean don't care
| '*'
| unsigned

sndrcv:

'snd'
operation: send message
| 'rcv'
operation: receive message

reqrel:

'req'
operation: request semaphore
| 'rel'
operation: release semaphore

task:

spec

spec:

module tmsname

module:

modname '|'
varname:

strunkomp:
modname:
tmsname:
procname:

IDENTIFIER

unsigned:

LONG

stoptasks:	
stoptrace stoplist	
stoptrace:	defines 'stop trace' action
<empty>	don't stop traces
| 'stoptrace' '\0'	stop all traces
stoplist:	
'nothing' '\0'	don't stop any tasks
| tasklist	stop specified tasks
tasklist:	
'all' '\0'	all tasks
| tasks	
tasks:	list of tasks
task '\0'	
| task '\0' tasks	

NOTE:

The character '\0' denotes the end of the current string in the string array. All subsequent characters are located in the next string until the next '\0' and so on. The string array is terminated with a NULL pointer.

DESCRIPTION:

dbreak_s5() sets a breakpoint in the monitored node program. If an error occurs *dbreak_s5()* returns an error value in **except* and exits without any cleanup. This means that the state of the monitors is undefined when an error occurs during breakpoint definition.

If the specified predicate becomes true, the tasks that are defined in *stoptasks* are stopped. In addition, a currently active trace can be stopped, if 'stoptrace' is specified in *stoptasks*. To inform the caller of *dbreak_s5()* about the occurrence of the breakpoint, the function pointed to by *handler* is called.

For more information about breakpoint predicates, see the DETOP User's Guide.

BUGS:

The 'stoptrace' action currently doesn't do anything.

Only one breakpoint can be defined. A subsequent definition will clear any previously defined breakpoint.

NAME:

cbreak_s5() - clear breakpoint

SYNOPSIS:

void cbreak_s5(except)

Errmsg *except;	Pointer to return value

DESCRIPTION:

cbreak_s5() clears the currently active breakpoint. If no breakpoint exists *cbreak_s5()* returns without any action.

NAME:

mbreak_s5() - send masterbreak to cube

SYNOPSIS:

void mbreak_s5 (except)

Errmsg *except;	Pointer to return value

DESCRIPTION:

mbreak_s5() stops all tasks running on the monitored subcube immediately.

NAME:

dtrace_s5() - define a trace

SYNOPSIS:

void dtrace_s5(except,tracespez,tracenr):

Errmsg *except;	Pointer to return value
char *tracespez;	string containing the specification of the trace that is to be defined
int *tracenr;	pointer to a variable that contains the trace number when *dtrace_s5()* returns successfully. The trace number is needed as input for subsequent calls to *ctrace_s5()* and *traceb_s5()*.

STRING SYNTAX:

tracespez:	
'exectr' exectrace	execution trace
| 'dtr' datatrace	data trace
| 'exchtr' exchtrace	exchange trace
exectrace:	execution trace needs only specification of a task
'#' task	
datatrace:	data trace
task '\' locvar	locvar and globvar syntax is as in *dbreak_s5()*
task '|' globvar	
exchtrace:	exchange trace
'mail' mailbox sndrcv messtask	trace all operations (send/receive as specified by sndrcv) on the mailbox
| 'sema' semaphore reqrel unittask	the same for semaphores
| 'task' task sndrcv 'mail'	trace all mailbox operations (as specified by sndrcv) executed by the given task
| 'task' task reqrel 'sema'	the same for semaphores
messtask:	this part of the syntax is maintained for compatibility reasons only and doesn't have any function
'mess'	
| 'task'	
unittask:	this part of the syntax is maintained for compatibility reasons only and doesn't have any function
'unit'	
| 'task'	

task:
mailbox:
semaphore:
spec see *dbreak_s5()*

DESCRIPTION:

dtrace_s5() defines a trace of the monitored node program. This means it sets up the monitor of the appropriate node so that the monitor records events defined by *tracespez* in its trace buffer. See the DETOP Reference Manual for more information about traces.

If the call is successful, **except* contains OK and **tracenr* contains the number of the trace, otherwise **except* contains an error value as defined in HLTL.h. The trace number is used to identify the trace and is passed to the *ctrace_s5()* and *traceb_s5()* procedures as a parameter.

NAME:

ctrace_s5() - clear a trace

SYNOPSIS:

void ctrace_s5 (except,trace_no)

Errmsg *except;	Pointer to return value
int trace_no;	The number of the trace that has to be deleted. This number has been returned by *dtrace_s5()* when the trace was defined.

DESCRIPTION:

ctrace_s5() clears the trace specified by *trace_no*. Clearing the trace means stopping the monitoring and deleting the trace buffer.

NAME:

traceb_s5() - get contents of a trace's buffer

SYNOPSIS:

void traceb_s5 (except,trace_no,trace_bed_out)

Errmsg *except;	Pointer to return value
int trace_no;	The number of the trace whose trace buffer should be returned. This number has been returned by *dtrace_s5()* when the trace was defined.
Trace_bed ** trace_bed_out;	Pointer to a variable that contains a pointer to a *Trace_bed* structure when *traceb_s5()* returns successfully. Do NOT free any part of this structure!

DESCRIPTION:

traceb_s5() returns the trace buffer of the trace defined by *trace_no*. For efficiency reasons the buffer is not copied into a separate data structure, but a pointer to an internal data structure of the HLTL is returned. Do not change any part of this data structure!

See the DETOP User's Guide for detailed information on the data that are stored in the trace buffer.

NAME:

ltrace() - get a list of all active traces

SYNOPSIS:

void ltrace (except,traces)

Errmsg *except;	Pointer to return value
Trace_bed ** traces;	Pointer to a variable that contains a pointer to the first element of a linked list of *Trace_bed* structures when *ltrace()* returns successfully. Do NOT free any part of this structure!

DESCRIPTION:

ltrace() is intended to return a list of all active traces. However, it just returns a pointer to an internal data structure of the HLTL. It is the task of the calling program to extract the appropriate information from this data structure. The caller must not change any part of this structure!

2 Go and Single Step Commands

NAME:

go_s5() - start tasks

SYNOPSIS:

void go_s5(except,task_list)

Errmsg *except;	Pointer to return value
char *task_list[];	Array of pointers to strings that contain the list of tasks to be started.

STRING SYNTAX:

task_list:

tasklist	see *dbreak_s5()*

DESCRIPTION:

go_s5() starts or resumes the tasks of the node program that are specified in *task_list*.

NAME:

sstep_s5(), pstep_s5() - execute a task in single step mode

SYNOPSIS:

void sstep_s5(except,task_bez,no_of_steps,stop_msg)

void pstep_s5(except,task_bez,no_of_steps,stop_msg)

Errmsg *except;	Pointer to return value
char *task_bez;	this string contains the definition of the task that has to be single-stepped.
int no_of_steps;	the number of singe-steps to be performed
char ***stop_msg;	Pointer to a variable that points to a string array when *sstep_s5()* or *pstep_s5()* returns successfully. The string array contains the current calling sequence and the current line number of the single-stepped task. The memory for this structure is allocated by *sstep_s5()* or *pstep_s5()* and should be freed by the caller after usage.

STRING SYNTAX:

task_bez:

task — see *dbreak_s5()*

stop_msg:

callseq lineno

callseq:

module '\0' proc '.' '\0'
| module '\0' proc '.' '\0' callseq

lineno:

'<' unsigned '>' '\0'

NOTE:

The character '\0' denotes the end of the current string in the string array. All subsequent characters are located in the next string until the next '\0' and so on. The string array is terminated with a NULL pointer.

DESCRIPTION:

sstep_s5() and *pstep_s5()* perform single step execution of one node task. The difference between these procedures is that with a *sstep_s5()* call the node tasks steps into procedures while with a *pstep_s5()* call the node task will execute procedure calls as a single step. The string array returned in *stop_msg* contains the current calling sequence and source line number of the task after the specified number of steps has been performed. Again, refer to the DETOP User's Guide for further information.

BUGS:

If the node tasks runs into an infinite wait condition (for example due to a *recmsg()* call) *sstep_s5()* and *pstep_s5()* does not return.

3 Inspect and View Commands

NAME:

inspect_s5() - get state information of a MMK object

SYNOPSIS:

void inspect_s5(except,tms,obj_bez,desc)

Errmsg *except;	Pointer to return value
Type_tag tms;	defines the type of MMK object to be inspected. Legal values are *T_task, T_mail* and *T_sema*.
char *obj_bez;	a string containing the specification of the MMK object.
Obj_Desc *desc;	pointer to an *Obj_Desc* structure where *inspect_s5()* places its result when returning successfully.

STRING SYNTAX:

obj_bez:	
spec	see *dbreak_s5()*

DESCRIPTION:

inspect_s5() returns a data structure containing information about the current state of the specified MMK object. See the DETOP User's Guide and the description of data structures at the end of this chapter.

NAME:

inspect_node_s5() - get all objects of given type on a given node

SYNOPSIS:

void inspect_node_s5 (except, node, tms, obj_list)

Errmsg *except;	Pointer to return value
Node node;	the number of the node to be inspected
Type_tag tms;	can be either *T_task, T_mail* or *T_sema*
char ***obj_list;	Pointer to a variable that points to a string array when *inspect_node_s5()* returns successfully. This array contains the names of all MMK objects that are located on the specified node and are of the required type. The memory for this structure is allocated by *inspect_node_s5()* and should be freed by the caller after usage.

STRING SYNTAX:

obj_list:

spec '\0' — see *dbreak_s5()*
| spec '\0' obj_list
| '<no|object>' '\0'

NOTE:

The character '\0' denotes the end of the current string in the string array. All subsequent characters are located in the next string until the next '\0' and so on. The string array is terminated with a NULL pointer.

DESCRIPTION:

inspect_node_s5() returns a string array that contains the names of all MMK objects that match the type given by *tms* and are located on node *node.*

NAME:

view_s5() - get a list of all MMK objects with given type

SYNOPSIS:

void view_s5 (except, tms, obj_list)

Errmsg *except;	Pointer to return value
Type_tag tms;	defines the type of the MMK objects to be listed. Legal values are *T_task, T_mail* and *T_sema*.
char ***obj_list;	Pointer to a variable that points to a string array when *view_s5()* returns successfully. These strings contain the names of all currently available MMK objects of the selected type. The memory for this structure is allocated by *view_s5()* and should be freed by the caller after usage.

STRING SYNTAX:

see *inspect_node_s5()*

DESCRIPTION:

view_s5() is similar to *inspect_node_s5()*, but returns the names of all MMK objects that are of type *tms* regardless of their location on the cube.

NAME:

view_nodes_s5() - get a list of all active nodes

SYNOPSIS:

void view_nodes_s5(except, active_nodes)

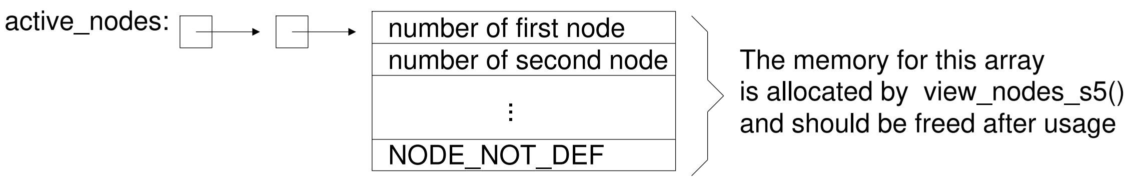
Errmsg *except;	Pointer to return value
Node **active_nodes;	Pointer to a variable that points to a node array when *view_nodes_s5()* returns successfully. The last element in this array has the special value NODE_NOT_DEF. The memory for this array is allocated by *view_nodes_s5()* and should be freed with *free(*active_nodes)* after usage.

DESCRIPTION:

view_nodes_s5() returns the numbers of all nodes that are in use by the monitored node program.

Structure of **active_nodes:

Node **active_nodes;



NAME:

view_symbol() - get all symbols of a given class

SYNOPSIS:

void view_symbol (except,class,what,Aview_return)

Errmsg * except;	Pointer to return value
view_symbol_class class;	defines the class of symbols returned via Aview_return. Can be either MODULE, FUNKTIONEN or VARIABLEN
char ** what;	If *class* == MODULE this parameter is a don't care. If *class* == FUNKTIONEN *what[0]* must point to a string containing the name of the module whose functions are to be returned If *class* == VARIABLEN *what[0]* must point to a string containing the name of a module. If *what[1]* == NULL, all global variables are returned, if *what[1]* is not NULL, it must point to a string containing the name of a function within the specified module. In this case all local variables of that function are returned.
char * Aview_return;**	Pointer to a variable that points to a string array when *view_symbol_s5()* returns successfully. This string array contains all symbolic names that are selected by class and what. The memory for this structure is allocated by *view_symbol_s5()* and should be freed by the caller after usage.

STRING SYNTAX:

Aview_return:

modlist
| proclist
| varlist

modlist:

modname '\0'
| modname '\0' modlist

proclist:

procname '\0'
| procname '\0' proclist

varlist:

varname '\0'
| varname '\0' varlist

NOTE:

The character '\0' denotes the end of the current string in the string array. All subsequent characters are located in the next string until the next '\0' and so on. The string array is terminated with a NULL pointer.

DESCRIPTION:

view_symbol_s5() allows access to the symbol table information of the UTOL. It returns a list of all symbols that match the type in the *class* parameter. To get a list of all modules, set *class* to MODULE and leave *what* undefined. If you want a list of all functions in a given module, set *class* to FUNKTIONEN and *what[0]* to the module's name. Finally, to get a list of all variables in a function, *class* must be VARIABLEN, *what[0]* must contain the name of the module and *what[1]* must contain the function's name. You can set *what[1]* to NULL to get a list of all global variables in the module.

4 Examination of Variables

NAME:

display_s5() - get contents of a variable

SYNOPSIS:

void display_s5(except,task_bez,var_bez,val_list)

Errmsg *except;	Pointer to return value
char *task_bez;	this string contains the definition of the task that defines the environment for the display. Syntax is shown below.
char *var_bez;	a string containing the specification of the variable to be displayed. Syntax is shown below.
Valel **val_list;	pointer to a variable that contains the pointer to an array of *Valel* structures when *display_s5()* exits successfully. The memory for the *Valel* structure is allocated by *display_s5()* and should be freed with *free(*val_list)* after usage.

STRING SYNTAX:

task_bez:

'#' task	if global variable is to be displayed
| task	if local variable is to be displayed

var_bez:

globalvar	global variable
| localvar	local variable
| conditional	local variable within current scope

globalvar:

module conditional

localvar:

dynlink ':' conditional

dynlink: dynlink is the complete dynamic path to the desired incarnation of the procedure containing the variable

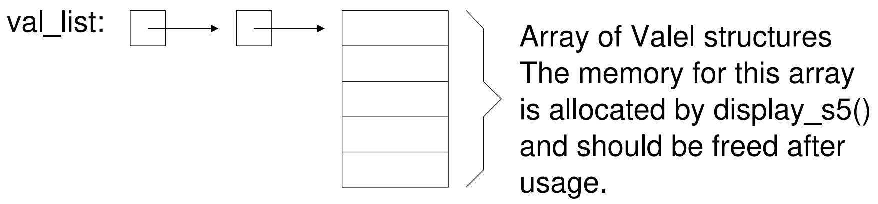
module proc
| module proc '/' dynlink

DESCRIPTION:

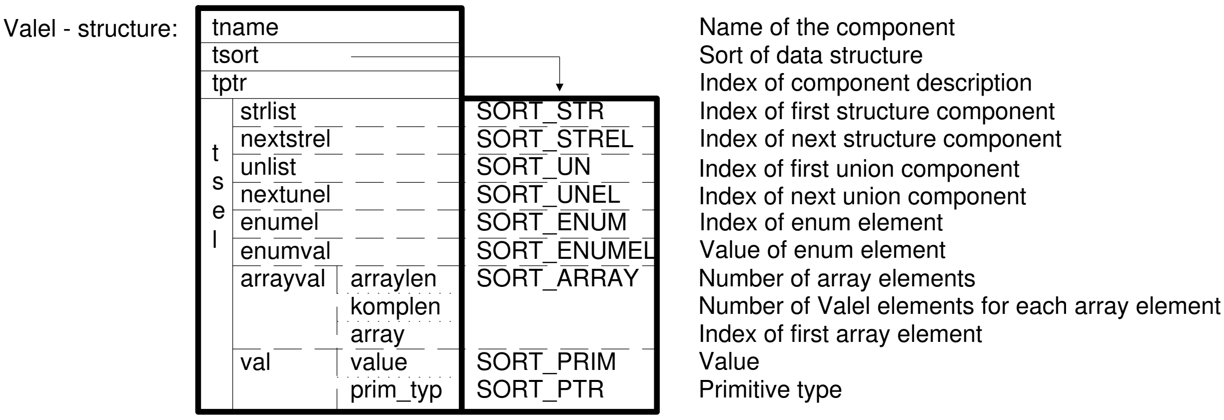
display_s5() determines the value of the variable specified by *var_bez* in the context of the task given by *task_bez*. The value is returned in a data structure that allows to determine the structure of that value (the variable can be of any C type). Please refer to the DETOP User's Guide for more details on how to define the variable.

Structure of **val_list:

Valel **val_list;

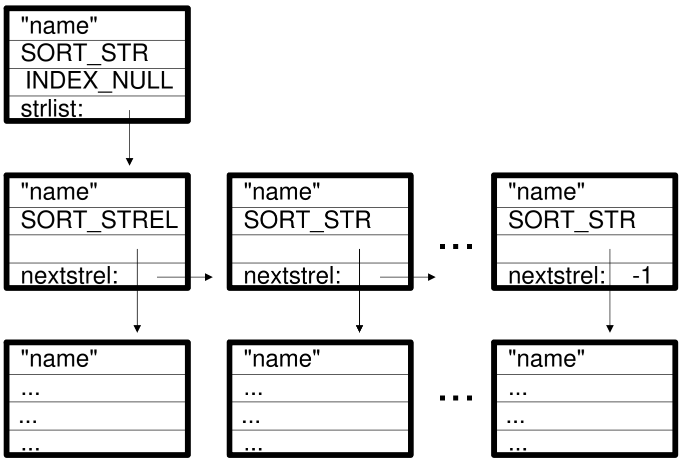


Structure returned by display_s5():



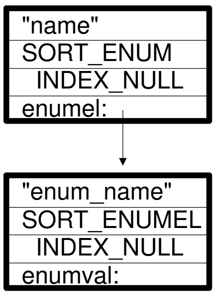
Examples:

Valel-array for a structure (or union):

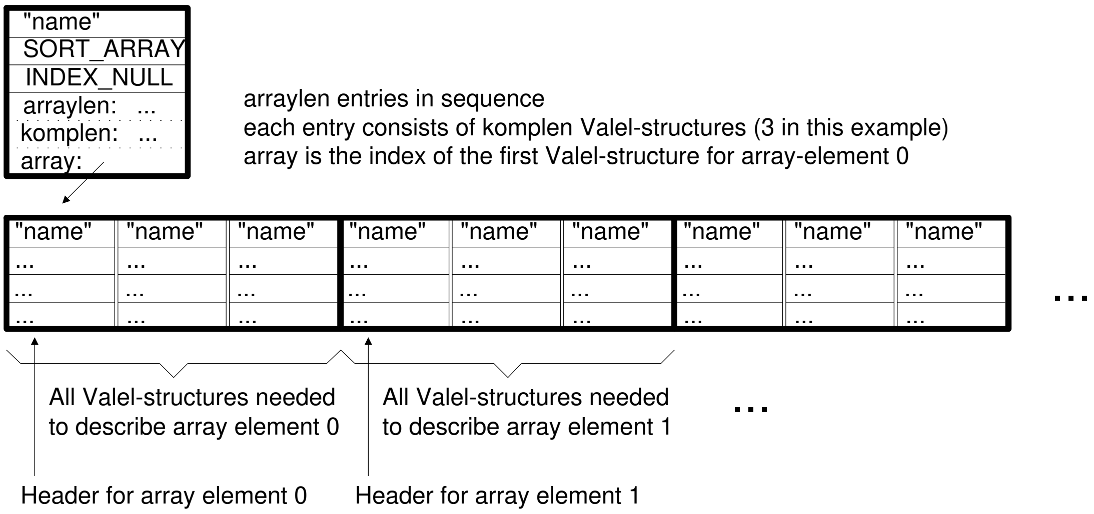


These Valel-elements may be of any type, thus this scheme is recursive

Valel-array for an enum variable:



Valel-array for an array:



Note:

The pointers shown in this drawing are actually indices in the Valal-array

NAME:

modify_s5() - change contents of a variable

SYNOPSIS:

void modify_s5(except,task_bez,modify_op)

Errmsg *except;	Pointer to return value
char *task_bez;	this string contains the definition of the task that defines the environment for the modify operation. Syntax is shown below.
char *modify_op;	this string contains a C-statement that is interpreted by *modify_s5()*. It usually contains some sort of assignment. The syntax is shown below.

STRING SYNTAX:

task_bez:

task	(see *dbreak_s5()*)

modify_op:

statement	(see C-syntax)

DESCRIPTION:

modify_s5() takes a C statement and the name of a node task as arguments and executes this statement in the current context of the task in an interpretive manner. Because essentially only assignments are allowed as statements, *modify_s5()* is mainly used to modify the contents of a variable.

NAME:

type_s5() - determine a variable's type

SYNOPSIS:

void type_s5(except,var_bez,typ_list)

Errmsg *except;	Pointer to return value
char *var_bez;	a string containing the specification of the variable whose type is to be determined.
Typel **typ_list;	pointer to a variable that contains the pointer to an array of Typel structures when *type_s5()* exits successfully. The memory for the *Typel* structure is allocated by *type_s5()* and should be freed with *free(*typ_list)* after usage.

STRING SYNTAX:

var_bez:
module conditional (see *dbreak_s5()* and C-syntax)
| module procname ':' conditional

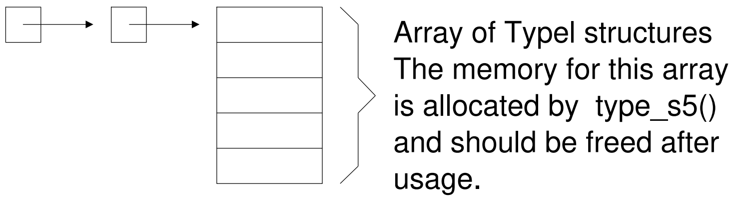
DESCRIPTION:

type_s5() is similar to *display_s5()* with the difference that not the variable's value but its type is returned. Note that no task definition is needed and the syntax of the variable description is more simple as in the case of *display_s5()*.

Structure of **typ_list:

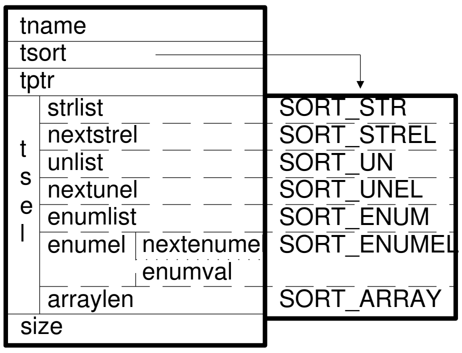
Typel **tpy_list;

typ_list:



Structure returned by type_s5():

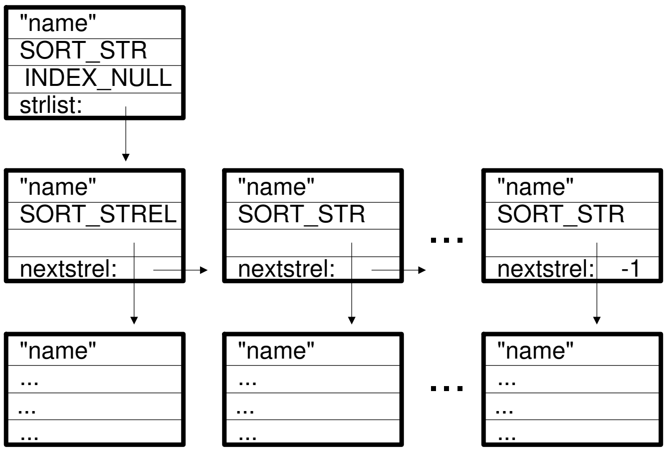
Typel - structure:



Name of the component
Sort of data structure
Index of component or pointer type description
Index of first structure component
Index of next structure component
Index of first union component
Index of next union component
Index of first enum element
Index of next enum element
Numerical value of enum element
Array length
Size of one element of this type in bytes

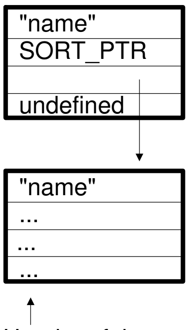
Examples:

Typel-array for a structure (or union):



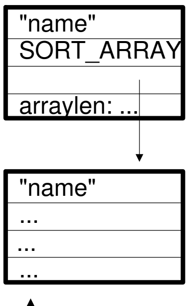
These Typel-elements may be of any type, thus this scheme is recursive

Typel-array for a pointer type:



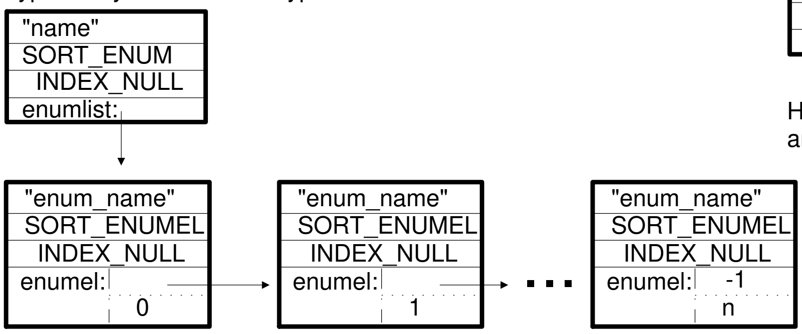
Header of the type-description of *pointer

Typel-array for an array type:



Header of the type-description of array elements

Typel-array for an enum Type:



Notes:

The pointers shown here are actually indices in the Typel-array
The size component is not shown in this drawing

5 Performance Measurements

NAME:

pa_define() - define a performance measurement

SYNOPSIS:

Tool_token pa_define (pa_type, node1, node2, s_obj1, s_obj2, program_part, granularity,except)

Pa_type pa_type;	Specifies the type of measurement that is defined
Node node1;	The first node that is involved in this measurement
Node node2;	The second node that is involved in this measurement
char* s_obj1;	The first MMK object that is involved in this measurement
char* s_obj2;	The second MMK object that is involved in this measurement
char* program_part;	Currently not used
Interval granularity;	The resolution of the measurement in ms.
Errmsg *except;	Pointer to return value

STRING SYNTAX:

s_obj1:

s_obj2:

NULL	i.e. NULL pointer
| spec	task, mailbox or semaphore spec. (see *dbreak_s5()*)

DESCRIPTION:

pa_define() is used to define a performance measurement on the monitored node program. The measurement is not started until *pa_start()* is called. The measurements shown in the following table can be defined:

pa_type	node1	node2	obj1	obj2	Description
TASKRUNNING	*	*	task	NULL	CPU time of task
SNDMESSAGE	*	*	mbox	task	KByte/s sent from mbox to task
	*	*	mbox	NULL	KByte/s sent from mbox to all tasks
RECMESSAGE	*	*	mbox	task	KByte/s received by mbox from task
	*	*	mbox	NULL	KByte/s received by mbox from all tasks
WAITSEND	*	*	task	NULL	time task is waiting to send a message
	*	*	task	mbox	time task is waiting to send a message to mbox
T_WAITREC	*	*	task	NULL	time task is waiting to receive a message, measured on task's node
	*	*	task	mbox	time task is waiting to receive a message from mbox, measured on task's node
M_WAITREC	*	*	mbox	task	time task is waiting to receive a message from mbox, measured on mbox's node
	*	*	mbox	NULL	time all tasks are waiting to receive a message from mailbox, measured on mbox's node
T_P_OPERATION	*	*	task	NULL	time task is waiting to request a semaphore, measured on task's node
	*	*	task	sema	time task is waiting to request units from sema, measured on task's node
S_P_OPERATION	*	*	sema	NULL	time all tasks are waiting at sema, measured on sema's node
	*	*	sema	task	time task is waiting at sema, measured on sema's node
READYQUEUE	node	*	NULL	NULL	Length of ready queue on node
QUEUE_ON_MBOX	*	*	mbox	NULL	time tasks are waiting at mbox
QUEUE_ON_SEMA	*	*	sema	NULL	time tasks are waiting at sema
QUEUE_ON_RPC	node	*	NULL	NULL	time tasks are waiting for a RPC on node
SEND_BYTES	node1	*	NULL	NULL	KByte sent from node1 to all other nodes
	node1	node2	NULL	NULL	KByte sent from node1 to node2
IDLE_TIME	node	*	NULL	NULL	idle time of node

*: NODE_NOT_DEF

**: all times are mearured in µs

The *Tool_token* returned by pa_*define()* is used to address the measurement in the *pa_start(), pa_remove()* and *pa_read()* procedures.

If the call is successful, **except* contains OK, otherwise **except* contains an error value as defined in HLTL.h and the special *Tool_token* NO_TOOL_TOKEN is returned.

NAME:

pa_read() - read the next performance value

SYNOPSIS:

Pa_value pa_read(pa_token,except)

Tool_token pa_token;	The token of the measurement to be read. This token was returned by *pa_define()* when the measurement was defined.
Errmsg *except;	Pointer to return value

DESCRIPTION:

pa_read() is used to get the next performance value of the measurement specified by *pa_token*.

NAME:

pa_remove() - remove a measurement

SYNOPSIS:

pa_remove(pa_token,except)

Tool_token pa_token;	The token of the measurement to be deleted. This token was returned by *pa_define()* when the measurement was defined.
Errmsg *except;	Pointer to return value

DESCRIPTION:

pa_remove() removes the performance measurement specified by *pa_token*.

NAME:

pa_start() - start a measurement

SYNOPSIS:

pa_start(pa_tokens,except)

Start_list pa_tokens;	Pointer to a linked list of *Start_elem* structures defining all measurements to be started
Errmsg *except;	Pointer to return value

DESCRIPTION:

pa_start() starts the performance measurements specified in *pa_tokens*. *pa_start()* modifies the *start_time* components of the *Start_list* elements so that they point to a *timeb* structure containing the actual starting time of the measurement. These *timeb* structures exist as long as the measurement is running.

6 Initialization

NAME:

init_utol() - initialize the Universal Tool Layer

SYNOPSIS:

void init_utol(except,arg_anz,arg_vek)

Errmsg * except;	Pointer to return value
int arg_anz;	Should be set to *argc* of the *main()* routine
char **arg_vek;	Should be set to *argv* of the *main()* routine

DESCRIPTION:

init_utol() initializes the UTOL and the monitors. During the initialisation the files <nodeprog>.symb and <nodeprog>.global are read and the symbol tables and localizer lists are built up. In addition a list containing the start addresses of each C statement of the node program is sent to the monitors. This information is needed for breakpoints, tracing and single-stepping.

The name of the nodeprogram (above referenced as <nodeprog>) is extracted from the command line parameters via the *arg_vek* and *arg_anz* parameters.

init_utol() must be called exactly once before any of the other HLTL procedures can be used.

7 C-Syntax

Syntax for C-statements (*modify_s5()*) and C-expressions (*type_s5()*, *display_s5()*)

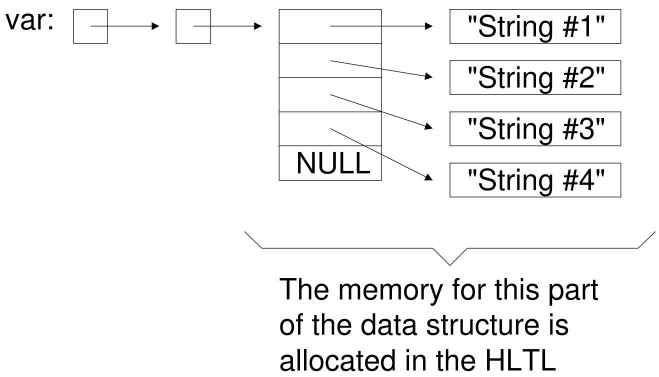
```
statement:
    opt_expression line_end

line_end:
    <empty>
    | ';'

opt_expression:
    <empty>
    | expression

expression:
    conditional
    | unary '=' conditional          (modify only)
    | unary '+=' conditional         (modify only)
    | unary '-=' conditional         (modify only)
    | unary '*=' conditional         (modify only)
    | unary '/=' conditional         (modify only)
    | unary '%=' conditional         (modify only)
    | unary '>>=' conditional        (modify only)
    | unary '<<=' conditional        (modify only)
    | unary '&=' conditional         (modify only)
    | unary '^=' conditional         (modify only)
    | unary '|=' conditional         (modify only)

conditional:
    bit_xor
    | conditional '|' bit_xor

bit_xor:
    bit_and
    | bit_xor '^' bit_and

bit_and:
    equality
    | bit_and '&' equality

equality:
    comparison
    | equality '==' comparison
    | equality '!=' comparison

comparison:
    shift
```

```
    | comparison '<' shift
    | comparison '<=' shift
    | comparison '>' shift
    | comparison '>=' shift
shift:
    addition
    | shift '>>' addition
    | shift '<<' addition
addition:
    multiplication
    | addition '+' multiplication
    | addition '-' multiplication
multiplication:
    unary
    | multiplication '*' unary
    | multiplication '/' unary
    | multiplication '%' unary
unary:
    primary
    | primary '++'                    (modify only)
    | primary '- -'                   (modify only)
    | '*' unary
    | '&' unary
    | '-' unary
    | '!' unary
    | '~' unary
primary:
    IDENTIFIER
    | CHAR
    | INT
    | LONG
    | DOUBLE
    | '(' expression ')'
    | primary '[' conditional ']'
    | primary '.' IDENTIFIER
    | primary '-' IDENTIFIER
```

8 Data Structures Used in the HLTL Interface

Structure of a string-array:

char ***var;



Obj_Desc structure:

```
typedef union {
        Task_Desc task;
        Mail_Desc mail;
        Sema_Desc sema;
} Obj_Desc;
```

typedef struct {

char *name;	Name of the task
unsigned priority;	NOT USED
Task_state state;	scheduling state of the task, see below
char **calls	pointer to a string-array containing the current calling sequence of the task
char *waiting_at;	Name of mailbox or semaphore the task is waiting at.
unsigned np;	If the task waits for sending to a mailbox,np is the address of the message. If the task waits to receive a message, np is 0.
unsigned cre_adr;	NOT USED
unsigned units;	If the task waits due to a semaphore request,units contains the number of units requested by the task.
unsigned last_ip;	The source line number of the current execution point of the task

} Task_Desc;

STRING SYNTAX:

name:

task — see *dbreak_s5()*

calls:

callseq lineno — see *sstep_s5()*

waiting_at:

spec — see *dbreak_s5()*
| '<no|object>'

Coding of wait conditions:

wait condition	waiting_at	**np**	**units**
doesn't wait	<no object>	don't care	don't care
receive from mailbox	spec	0	0
send to mailbox	spec	message pointer	0
request semaphore	spec	0	requested units

typedef struct {	
char *name;	Name of the mailbox
unsigned buf_total;	Size of message buffer
unsigned buf_free;	Number of free buffer entries
unsigned *cont;	Pointer to an array that contains the addresses of all messages in the mailbox
char **queue;	Pointer to a string array containing the names of all tasks waiting at this mailbox
} Mail_Desc;	

STRING SYNTAX:

name:

mail — see *dbreak_s5()*

queue:

'<no|objects>' '\0'
| tasks — see *dbreak_s5()*

typedef struct {

char *name;	Name of the semaphore
unsigned units;	Number of free semaphore units
char **queue;	Pointer to a string array containing the names of all tasks waiting at this mailbox

} Sema_Desc;

STRING SYNTAX:

name:

sema

queue:

'<no|object>' '\0'
| tasks

Trace_bed structure:

typedef struct trace_bed {

int trace_no;	the global trace number
char trace [TRACELEN];	the trace definition as supplied to *dtrace_s5()* when definig this trace
char loc_trace_no;	the local trace number on the node
Node node_no;	the number of the node that monitors this trace
Trace_mem trace_mem [TRACE_BUF_LEN];	the trace buffer. It is implemented as a circular buffer.
int begin_read;	index of the oldest trace buffer element that has not been read yet
int begin_write;	index of the first free element in the trace buffer.
Break_def parsed_trace;	a structure that contains the trace definition in a parsed form.
struct trace_bed * next_trace;	pointer to the *Trace_bed* structure of the next trace

} Trace_bed;

typedef struct {	
long lok_trace_no;	number of the trace this entry belongs to
union {	
long token;	Task-,Mailbox- or Semaphore-token
struct {	Task-,Mailbox- or Semaphore-name (exchange trace)
char mod_name[ID_LEN];	
char tms_name[ID_LEN];	
} name;	
} ta_se_mai;	
Simple_type value;	Value of the traced variable (data trace) or line number (execution trace) or message pointer (exchange trace on a mailbox) or semaphore units (exchenge trace on a semaphore)
long time;	time stamp
} Trace_mem;	

Start_list structure:

typedef struct _Start_list {	
Tool_token tool_token;	token of measurement that is to be started
struct _Start_list *next;	Pointer to next list element
struct timeb *start_time;	pointer to a timeb structure that is set up by the *pa_start()* call and is valid as long as the measurement is running
} *Start_list;	

Part 5:

The Distributed Monitoring System

Thomas Treml

The following chapters describe the concepts and implementation of the TOPSYS monitoring system. The entire code of the monitor described here is linked with the application and runs on the iPSC/2. Communication with the tools, which are treated in the previous parts, is accomplished by message passing. First the concepts of predicates and actions are described. This is the framework for the following chapters, which deal with the implementation of the concepts. This part is augmented by three appendices. These appendices list the functionality of the interfaces between the UTOL (on the host workstation) and the monitor, an interface layer within the monitor (to seperate parts which could be handled in hardware) and the interface to the MMK. The implementation is only described for the software monitor.

1 The Concept of Predicates and Actions

1.1 Predicates

One of the main tasks of the TOPSYS monitoring system is to detect the satisfaction of predicates. We distinguish between three kinds of predicates: execution predicates, data predicates and exchange predicates. These predicates are called simple predicates, because they can be detected by the means of a single monitor. The simple predicates can be combined sequentially or by of the OR operator to form complex predicates. If the simple predicates, which form the complex one, are spread over several nodes, node local monitoring is not sufficient for recognizing this complex predicate. In this case there has to be communication between the node local monitors. Examples for simple predicates are:

task t1 enters/leaves procedure alpha

task t2 executes instruction #35

variable x gets value 4

task t1 sends a message to mailbox m2

These simple predicates can be combined, for example:

task t1 enters procedure alpha (p1) and then assigns 4 to variable x (p2) or sends a message to mailbox m2 (p3)

p1 -> p2 v p3

Execution predicates are specified by the code address, which should be supervised, and optionally a repetition counter and a task. The repetition counter determines how many times the code at the specified code address has to be executed before the predicate turns true.

Data predicates can be specified for local or global variables. For local variables it is possible to specify a certain incarnation of the procedure. Read or write accesses can be monitored. If the access direction is write, then it is possible to specify a value. The specification of a task is mandatory.

For exchange predicates you can specify the involved MMK objects (task, mailbox, semaphore), the operation and some parameters of this operation.

1.2 Actions

If a user defined predicate (simple or complex) is fulfilled, the monitor system should do some action specified beforehand. There are two classes of actions. Actions which are executed due to fulfilled predicates are called conditional actions. Actions which are executed to answer a user request are called unconditional actions. The first class consists of actions like stopping tasks (breakpoint), tracing data or incrementing/decrementing counters. The second class is mainly used to inspect the state of the user program (e.g. display the contents of variables, inspect the state of tasks, mailboxes and semaphores). Actions may also be spread over several nodes.

1.3 Assembly of Complex Predicates and Actions

OR combined predicates are put together in a so called OR block. For every predicate it is possible to specify a list of actions, which should be performed when the predicate is fulfilled (see fig. 1.1).

predicate 1	action list 1
predicate 2	action list 2
. . .	. . .
predicate n	action list n

Fig. 1.1: OR block

Generally there are several of these OR blocks chained in a linear list (see fig. 1.2).

The predicates in the first OR block are supervised. One of the actions in the action list is called block switch action. This action switches supervision to another OR block and realizes hereby the sequential operator ->. You can consider this structure to be a deterministic finite automaton of Mealy type. The predicates are the input characters, the OR blocks are the states of the automaton and the actions are the output characters. The state transition and output character depend on the current state (OR block) and the fulfilled predicate.

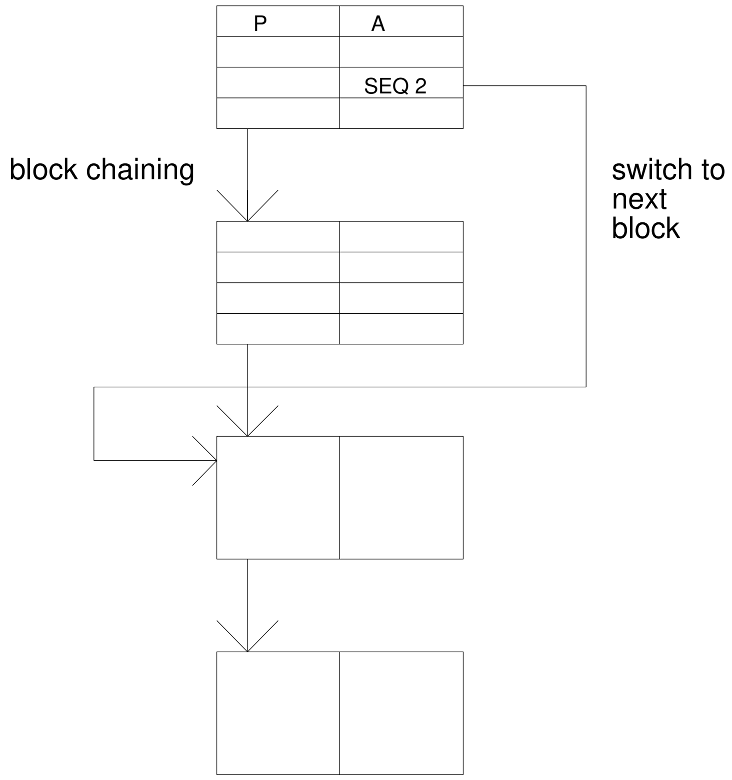


Fig. 1.2: List of OR blocks

1.4 Global Supervision by Distribution and Crosstriggers

It was mentioned above, that complex predicates may concern several nodes. The actions associated with predicates also may be spread over several nodes. Therefore it is necessary to distribute the components of the complex predicates to the nodes, where they should be satisfied. Generally these are the nodes, where the tasks involved in the predicate are executed.

It is not enough to distribute the predicates and actions. There must also be a means of communication between the node local monitors, to detect the satisfaction of complex predicates, which are spread over several nodes, and to initiate actions at other nodes. For this purpose we use the so called crosstriggers (CT). Local monitors can send crosstriggers to other monitors or receive crosstriggers from other monitors. Sending a crosstrigger is considered to be an action of the sending monitor in reaction to a fulfilled predicate. A received crosstrigger is treated like a fulfilled predicate and the associated action is handled the same way. The necessary crosstriggers are determined by the host, which distributes them together with the user defined predicates and actions. For example: The user wants to stop all tasks, when task t1 (residing on node 0) or task t2 (residing on node 1) send a message. He will specify:

p1 v p2 | break

The software will translate and distribute this:

node 0:

If p1 is fulfilled send crosstrigger CT1 to all nodes and stop all tasks on this node

If CT1 is received stop all tasks on this node

node1:

If p2 is fulfilled send crosstrigger CT1 to all nodes and stop all tasks on this node

If CT1 is received stop all tasks on this node

1.5 Local Monitoring

Events

The predicates mentioned above can not be supervised directly but have to be composed of so called events. Events we are interested in are the execution of instructions defined beforehand (execution events), the access to memory locations defined beforehand (data events), crosstriggers (crosstrigger events) and certain values of counters (counter events). These events have to be composed to form predicates. This is accomplished by a deterministic finite automaton. The input characters are the events. When the automaton reaches a final state the predicate is fulfilled. As an example the events and the automaton for the data predicate "task t1 writes to variable x in the 5th incarnation of procedure alpha" are described:

First of all it is necessary to determine whether task1 is running. This is accomplished by watching the (MMK kernel) variable which contains a pointer to the running task (data event) and the code address of the first instruction of the scheduler (execution event). Then we have to watch the entry and exit of procedure alpha (execution event). The event "variable x is written" (data event) is the last event to be watched. Combining them results in the deterministic automaton of figure 1.3.

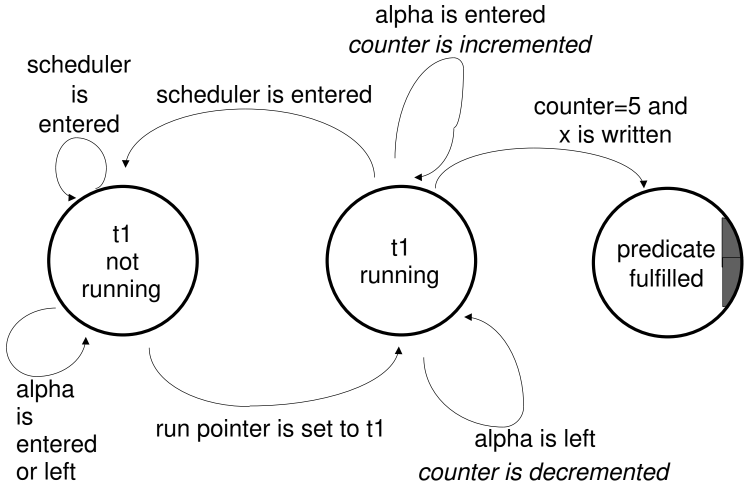


Fig. 1.3: DFA for a predicate

The automatons which detect the satisfaction of predicates and the structure for predicates and actions are composed to form one big automaton which does the whole supervision.

2 UTL and SW-monitor

2.1 Hierarchical Layer Model

In the hierarchical layer model mentioned in the introduction, you can find the monitors in the lower layers residing on the iPSC/2. Communication with the upper layers is defined by the S1 interface. The S1 interface is a command interface. For a detailed description of the available commands and their syntax see appendix A. Below the S1 interface there is the UTL (Universal Target Layer). This layer is responsible for communication with the host and performs actions which need no monitoring. These actions are for example inspecting tasks and mailboxes or changing the value of variables. The lowest layer is either realized by a software monitor or a hardware monitor. Choosing the appropriate monitoring technique is left to the user by linking his application with the MMK and one of the two monitors. The interface between UTL and the monitors is called S0 (see figure 2.1). It is a procedure oriented interface. You can find a listing with all procedure headers and their functional description in appendix B.

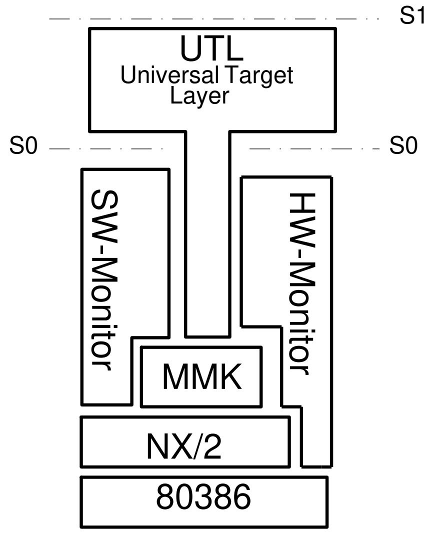


Fig. 2.1: TOPSYS layers on the iPSC/2

The implementation of the UTL and the monitors does not exactly reflect the layered model mentioned above. Instead of the layers there are two MMK tasks called PARSER and ANALYZER (see figure 2.2). Each of these tasks takes over some of the two layers' duties, whereby the PARSER does the biggest part of the UTL's job and the ANALYZER is mainly responsible for software monitoring. If the HW-monitor is used, the ANALYZER is replaced by a smaller version which is only invoked by interrupts generated by the HW-monitor. This smaller version stops tasks when a breakpoint is reached or does asynchronous communication with the host. Besides this the smaller version is used for single stepping through programs.

Commands from the host are sent to the PARSER. The PARSER investigates the received commands and executes them. Results of the commands are sent back to the host. The PARSER is waiting at a mailbox for host commands. On arrival of a command it executes it and waits again at the mailbox.

The ANALYZER does event recognition. Recognized events are composed to more complex events, which can be simple or complex predicates, and the appropriate action is executed. Fulfilled predicates are reported to the host. The ANALYZER is normally in the state “suspended”. On occurrence of a event it is resumed. After doing it’s job the ANALYZER suspends itself. Both tasks are running in an endless loop. There is shared memory for communication between the two tasks. Programming the monitor is done by the PARSER. It writes the necessary information for event recognition and actions into the shared memory or in the monitor registers, if the HW monitor is used. The ANALYZER does event recognition and execution of actions according to this information. Results of the actions are deposited in the shared memory too. After deposition of a result, the ANALYZER informs the host, which in turn sends a command to the PARSER to read this result from the shared memory. Besides the connection of the two monitor tasks via the shared memory, there is a connection via the events which should be recognized. At runtime the PARSER instruments the application program and activates monitoring hooks in the MMK respectively. Theese modifications invoke the ANALYZER when executed.

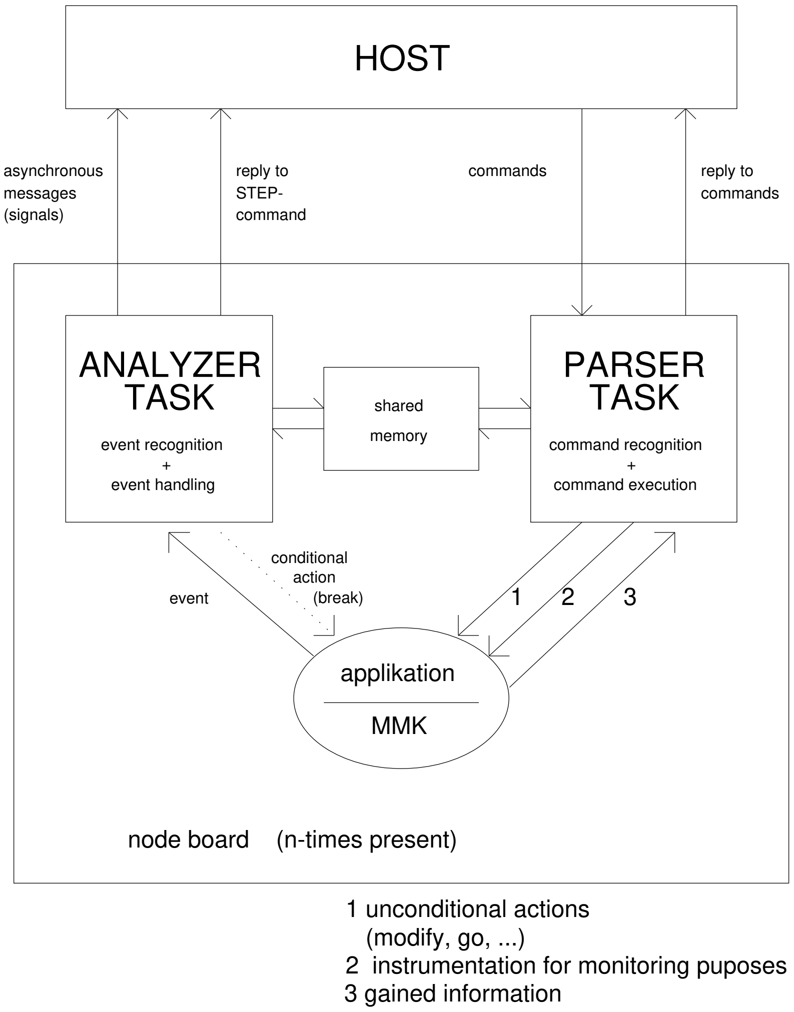


Fig. 2.2: The monitor tasks

For correct operation of the monitoring tasks it it necessary to priorize these tasks. Therefore the following priorities are given to the tasks:

highest priority	255	MMK init task; initialization at MMK start
	254	ANALYZER; it must not be interrupted by executing commands from the host or any other task
	253	PARSER; host commands are priorized to application tasks or MMK system tasks
	252	remote server tasks of the MMK
	100	application tasks
lowest priority	0	MMK idle task

2.2 Implementation

2.2.1 Host Communication

Communication with the host is done by invoking the communication services supplied by the NX/2 operating system. The nodes receive commands via the interrupt driven receiver routine of the NX/2 called “hrecv(...)”. Therefore on the arrival of a message a handler is invoked. The NX/2 handler invokes a MMK handler on completion (MMK handler concept). The MMK handler sends the message to the debug mailbox by invoking “sndmsg(...)”. This mailbox is a MMK object and the invoked function is a MMK system call. The parser fetches the message from the mailbox and executes the command which was conveyed in this message. As long as there is no message the parser is in the state “waiting”. The reply or the result of a command, respectively is sent to the host by the NX/2 service “isend(...)”. Beside replies to commands there are signals from the analyzer task, which are sent if a predicate becomes true on the node. These signals are sent with csend(..) (NX/2 call). At the host site the signals are changed to UNIX signals.

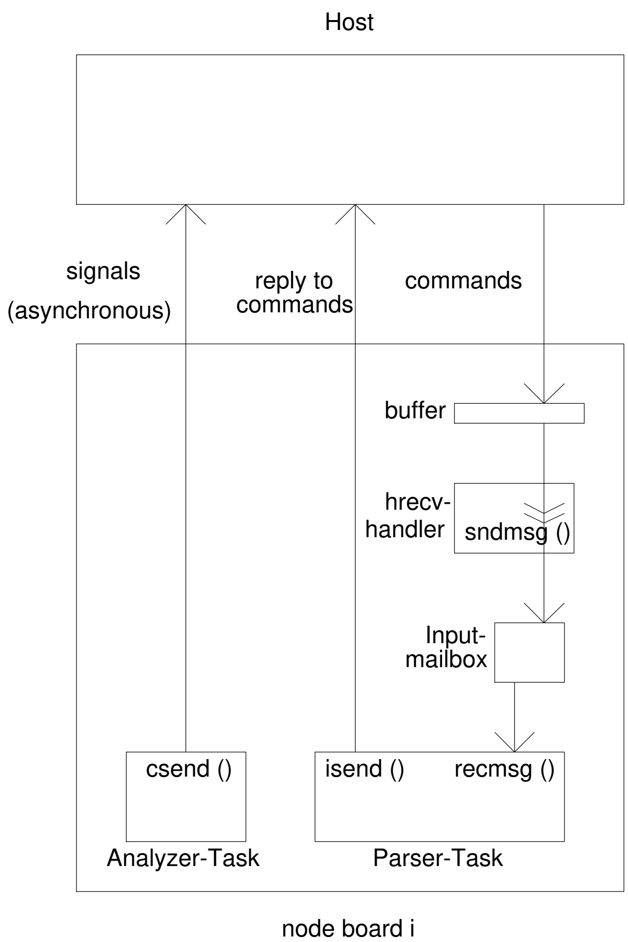


Fig. 2.3: Host communication

2.2.2 Event Recognition and Breakpoints

For recognition of execution events and data events the application's code is modified at the proper locations. The original opcode is hereby replaced with the breakpoint instruction (INT 3). This is done before starting the application or while it is running. When executing this instruction the program is interrupted and the invoked exception handler calls the analyzer task by executing the MMK system call restask(...). Nevertheless the instruction which was replaced by the "INT 3" opcode has to be executed. Therefore the ANALYZER rewrites the original opcode. Then the program counter is reset to the beginning of the instruction, the trap flag is set in the processor's status word and the task's priority is raised to a level which is higher than any other application task's priority. After this the ANALYZER suspends itself. Because of its high priority the interrupted task is resumed and one instruction is executed. Once more the task is interrupted. The reason is the single step interrupt (interrupt #1) caused by the previous set trap flag. This interrupt invokes a handler which resumes the ANALYZER. The instruction at this location is overwritten again with INT 3. Now the event is analyzed.

For monitoring execution predicates the breakpoint is set to locations in the object code, which correspond to instruction boundaries of the high level language source code. Monitoring entering and leaving of procedures is also done this way.

Recognition of data predicates needs a bigger amount of work, but uses the same technique of breakpoint setting. Generally the source or destination address of a memory access can not be ascertained by looking at the opcode. This is only possible at runtime immediately before the execution of the instruction, when all addressing parameters are fixed. Before runtime only the data type of the operation (byte, word, long) and the direction of access (reading, writing) can be ascertained. Therefore you have to pose a breakpoint instruction onto every instruction where a memory access with the desired data type and access direction occurs. On occurrence of the interrupt, the ANALYZER has to test whether the event corresponds to the expected address. This technique implies a lot of unnecessary program interrupts resulting in a drastically increasing runtime of the monitored application.

It is worth to mention, that after suspending the ANALYZER, not the interrupted task is continued, but another task with equal or higher priority. The interrupted task is chained at the end of the ready queue. This is caused by the preemptive priority driven round robin scheduling strategy of the MMK.

2.2.3 Monitoring Hooks

A lot of events, which are essential for monitoring, occur when MMK system calls are executed. To recognize these events the so called hook technique is used. The hooks are used for monitoring exchange predicates, doing exchange and performance traces and for all counters except the event counter.

For the hooks there is a special version of the MMK, which does function calls with the needed parameters for supervision. These function calls are realized by pointers to functions. If a function is not needed by the time, then only a dummy function is called, which returns immediately. Depending on the number of parameters there is a runtime delay in comparison with the “normal version” of the MMK even if only dummies are called.

When monitoring of a MMK function gets necessary, the pointer of the hook is set to point to a function called hook handler. These handlers are generally executed by the task, which did the system call. Therefore you have to take care that the hook handler is not suspended after a MMK time slice interrupt. This is done by placing the “disable()” command at the beginning of all hook handlers, if the time slice interrupt is not already inhibited by the MMK.

Treatment of exchange trace, performance trace and the counters is completely done within the handlers. This includes checking if the occurred event meets the trace definition and the trace itself. The same is done for the counters. First the event is checked for meeting the counter definition and then the counter is updated. There is no chance of getting the length of queues when the MMK code which does queue management is executed. Therefore the monitor has to have its own queue counters which are initialized once and then updated in the handlers.

To analyze exchange predicates, the handler function resumes the ANALYZER task. The analysis can not be done in the handler, because sometimes actions have to be triggered. If for example the action would be a break of the task, the handler would stop itself, for it is part of the task which is executed. Task switching inside the handler has to be done carefully. If the hook handler is executed during a critical region where the time slice interrupt is disabled, the handler must not make a task switch. Nonobservance of this fact would result in an unpredictable behaviour of the MMK.

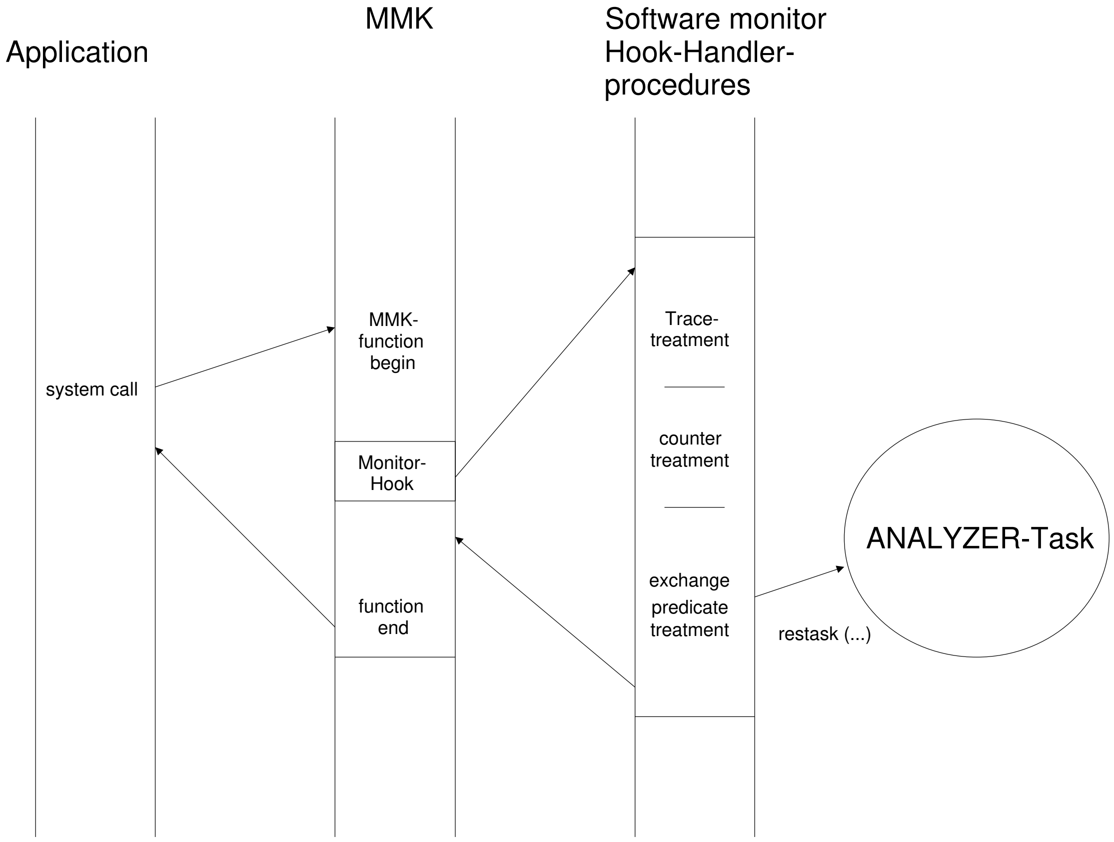


Fig. 2.4: Principle of Monitoring Hooks

You can find a list of all monitoring hooks in appendix C. In this appendix it is described where the hooks are inserted in the MMK, which functions of the monitor are supported (counters, exchange predicates, traces) and the number and types of parameters.

2.2.4 Changes in the MMK

Essential changes in the MMK are the hooks mentioned above. In addition there is a special monitoring hook for starting the monitor. This hook is placed in the MMK startup routine. The hook creates the two tasks PARSER and ANALYZER. Then communication with the host is initialized by starting a NX/2 handler (see Host Communication). The location where this startup hook is placed must not be changed, because the monitor relies on the order of the first entries in the MMK task list.

The hook insertion in the MMK source code was done by using the conditional translation mechanism of the C preprocessor:

```
#ifdef MONHOOKS
(* _mh_name)(...);
#endif
```

With this method it is possible to generate versions with and without the hooks. The branch targets of the hooks are initialized to a dummy handler:

```
_mh_name = _mh_none;
  void _mh_none()   { }
```

2.2.5 Changes to NX/2

Application, MMK and monitor are linked together to form one task (from NX/2's view), which is loaded on all nodes of the hypercube. This single NX/2 task runs on privilege level 3 of the 80386, which is the user level and lowest priority. NX/2 system calls are done this way: The application calls a library function. The library function does a software interrupt of a fixed number. The interrupt causes a priority change to privilege level 0, which is highest priority. The interrupt service routine calls the wanted NX/2 function via a so called interface function.

Essential for introducing a new NX/2 function is the introduction of a new interrupt number, which is distinct from all yet used numbers. In addition there has to be a library function and an appropriate interface function. This interface function calls the new NX/2 system service or an existing one.

2.2.5.1 Code Segment Access

Using the 80386 breakpoint interrupt makes it necessary to change the object code during runtime. The application program is not allowed to modify itself. When the application tries it a general protection fault is raised. The NX/2 has the necessary privilege level to do these modifications and there is a procedure which copies memory segments from arbitrary source locations to an arbitrary destination:

move(p,q,n);

p pointer to source data

q pointer to destination

n number of bytes to be copied

This procedure is not accessible by the user. Therefore the NX/2 had to be changed to make the procedure usable.

2.2.5.2 Microseconds Clock

There is a NX/2 system call for reading a clock. This clock has a resolution in the range of milliseconds. For monitoring purposes it is necessary to have a clock with a better resolution, to distinguish between moments when only some assembler statements are executed. This demands a clock with a resolution in the range of microseconds. Measuring time with the help of the software monitor is difficult because the monitor itself delays the application.

Two new NX/2 system calls have been implemented. The first one is for controlling the resolution of the clock and is called “set_base(time base)”. The time base has to be given in microseconds, whereby you must not fall below a minimum of 40 microseconds. The time base is initialized to 50 microseconds. The second system call is for reading the clock. The value returned is in units of the predefined time base. If you read the clock twice getting the values 10 and 15 and you have set the time base to 50 microseconds then the actual time lag is: (15-10)*50 microseconds= 250 microseconds.

The monitor’s clock uses the same hardware timer as the NX/2. The timer is a 8254 programmable interval timer, which is available on all node boards. Two counters of the timer are tied together to form a 16 bit counter. Counter 0 is triggered with a 1MHz clock. This counter counts backwards from 250 until it reaches 0. At this moment counter 1 is decremented and counter 0 starts over with 250. Counter one counts from four to zero and interrupts the processor to update the millisecond clock.

Time is computed by using the millisecond clock and the counters of the hardware timer:

microtime = millitime*(1000/time base)+

(4-counter_1)*(250/time base)+

(250-counter_0)/time base

Each counter value and the millisecond clock are converted to the time base and then summed up. The time base has to be supplied only for this conversion and is not used for programming the hardware timer. It is stored in a new introduced variable of NX/2.

The application reads the microsecond clock by using “uclock()”. Time base setting is done by the function “set_base(zb)".

2.2.6 Data Structures

2.2.6.1 The Hash Table

The data structure for storing the breakpoint information is a hash table. It is realized by an array of 256 pointers to linear lists (see figure 2.5). The elements of the list contain the necessary entries. The key function for hashing is the least significant byte of the program counter value, where the breakpoint should occur. Beside the pointer to the next element of the list, the entries contain the following information:

Code address in the object code

opcode at this address

type of breakpoint (what is the breakpoint used for); more than one entry is possible

pointer to a list, which contains pointers to the active execution predicates pointer to a list, which contains pointers to the active data predicates

pointer to a list, which contains pointers to the active data traces

The last two lists are used for watching the procedure incarnation of the data predicates and the data traces respectively

If the opcode does a memory access: access direction (reading/writing)

data type of the operation (byte/word/long)

When the monitor starts working it has to initialize this table. The command “initdeb” creates entries of code addresses for all procedure entries, procedure exits and all statement borders of the high level language used (C or Fortran). The type of the entries is “single step”, because this is the user’s view of program execution. Now the whole object code has to be investigated for the occurrence of instructions which do memory accesses. The addresses of these instructions is added to the table as “data breakpoints”. The remaining fields of the elements are filled when a breakpoint is activated.

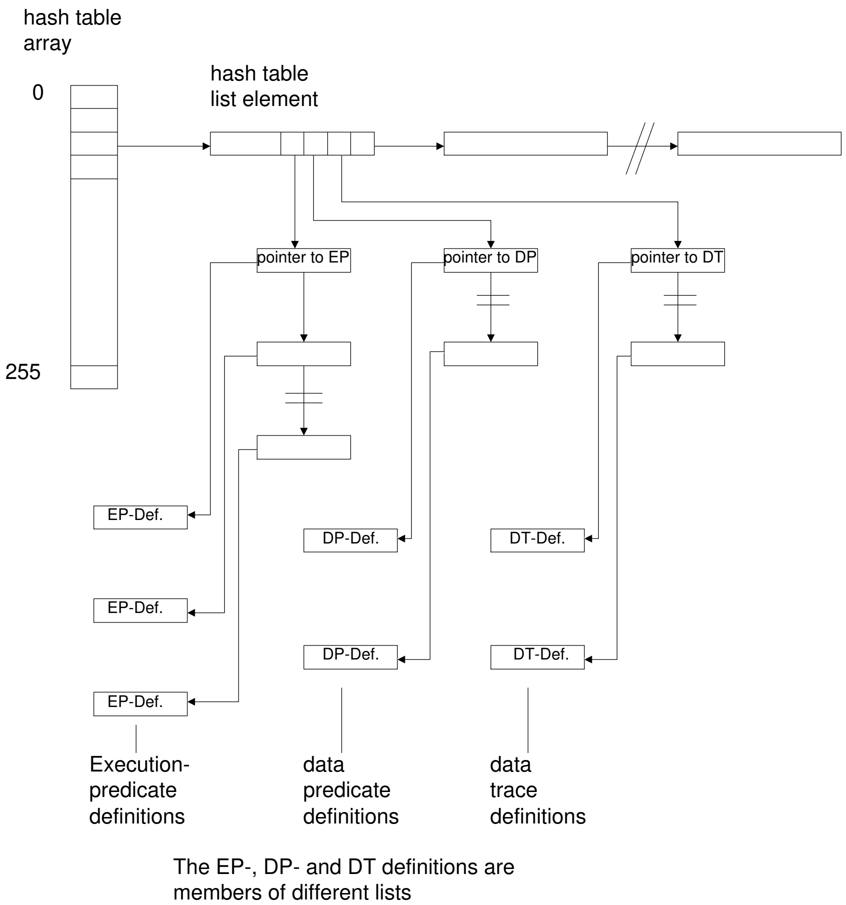


Fig. 2.5: Hash Table

2.2.6.2 The Table of Monitoring Hooks

The table of monitoring hooks is necessary for the administration of the hook handler functions. The monitoring hooks are used for supervision of exchange predicates, exchange traces, performance traces and counters. The table is organized as an array of elements which's entries are pointers to four lists. The lists contain pointers to the definition of the active XP, XT, PT and counters. Every array element of the table is assigned to a hook handler function. Therefore it is decideable within the hook handler what kind of monitoring has to be done.

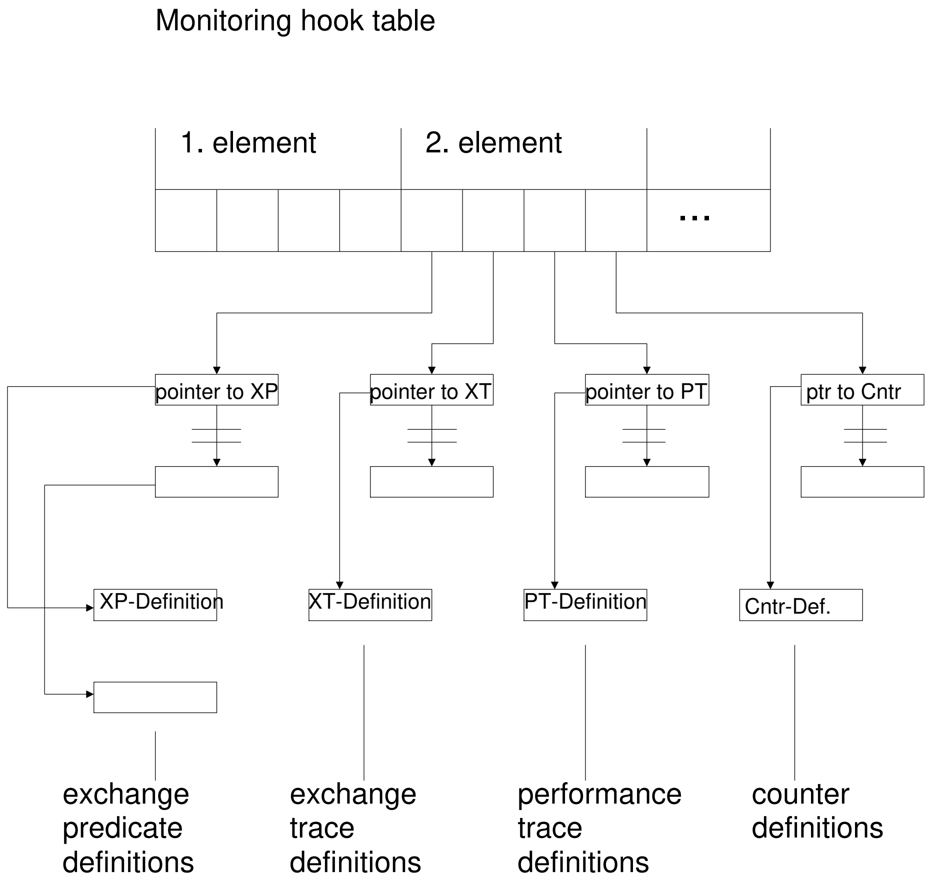


Fig. 2.6: Monitor Hook Table

2.2.6.3 The Data Breakpoint Counters

These counters are realized by an array of size 3x2. The elements are of type integer. The first row of counters reflects the number of active breakpoint definitions to realize data predicates. For each data type (B, W, L) there is a counter. The second row contains the values for the data predicates plus the values for data traces. A data breakpoint can be utilized by different data predicates and data traces at the same time. It is not necessary to deactivate the DTs and DPs at the same time. The data breakpoint must be cleared only when there is no DT or DP left, which utilizes this breakpoint.

2.2.6.4 The Trace Buffer

The trace buffer is used to deposit the data collected by traces. It is realized by an array of 5000 elements. This size is fixed but can be chosen arbitrary at compile time of the software monitor. The elements consist of five components which convey the following information:

1. Number of the trace definition which has caused this entry
2. MMK-object token 1
3. MMK-object token 2
4. Some data determined by the kind of trace
5. time stamp

The entries 1 and 5 are always present. The presence of the entries 2,3 and 4 depends on the kind of trace according to the S1 interface description. The trace buffer is treated as a ring buffer. If an overflow occurs the first entry is overwritten.

trace number	token 1	token 2	value	time

Fig. 2.7: Trace Buffer

2.2.6.5 The Structure for Storing Predicates and Actions

The structure for storing of predicates and actions is shown in figure 2.8.

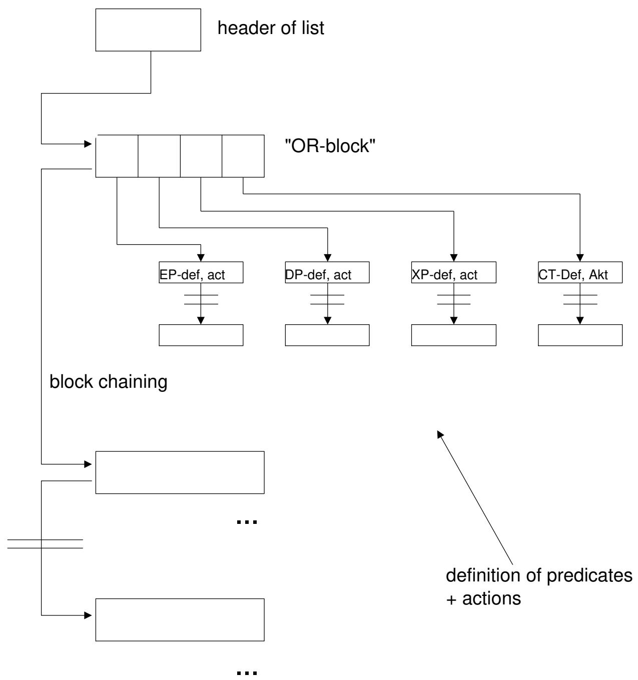


Fig. 2.8: Predicate Action structure

The structure is a direct representation of the chained OR blocks mentioned earlier. It is constructed after receiving a predicate-action command. For a description of the definition of predicates and actions see appendix A. The structure for execution predicates contains beside the variables for the EP definition a repetition counter. For data predicates there are also additional variables to hold the current incarnation of the procedure and to compute the absolute address of local variables. Beside these there is a variable which indicates whether a data breakpoint is set or only incarnation supervision is done.

2.2.6.6 Structure for Trace Definitions

For every kind of trace there is a list which stores all definitions of traces. In addition there is a compound variable whose components are pointers to these lists. So this variable is the head of each list. You can find the trace definitions in appendix A. Beside the trace definition the elements of the list contain the number of the trace and a variable which indicates whether a trace is active or not. The list elements of the data trace list have the same additional variables as the data predicate list. The lists are constructed by executing a trace command.

2.2.6.7 Structure for Counter Definitions

There is a list which chains all counter definitions. The elements are compound of the following components:

Number of the counter

Address of the MMK token variable for waiting queue monitoring at a selected object (optional)

Destination node number for monitoring of the message unit counter (optional)

Contents of the counter

Time of start

Time of last update

Length of the monitored waiting queue (optional)

2.2.7 Implementation of Predicate Supervision

2.2.7.1 Execution Predicates

For activating an EP a breakpoint is placed at the right address. Then a pointer to the EP definition is placed in the hash table according to the breakpoint address. By this method it is possible to access the EP definition in the current OR block directly, when the breakpoint is reached. To clear the EP the breakpoint instruction and the pointer have to be removed.

2.2.7.2 Data predicates

It was mentioned above that every instruction which does a memory access of the supervised types has to be superposed by a breakpoint instruction. The necessary locations can be found in the hash table. To set or reset a data breakpoint the whole hash table has to be scanned. If every entry in the hash table was augmented by a pointer to the DP definition, the hash table would need a huge amount of memory. Therefore all data definitions of the current OR block have to be scanned to determine the DP definitions which are concerned with the occurrence of a data breakpoint. If the DP is defined for a variable local to a procedure, it is possible to define the incarnation which should be supervised. In this case the breakpoints are set only when the expected incarnation is active. To watch the incarnations breakpoints are set at the beginning and the end of the procedure. In the elements of the hashtable for these two breakpoints a pointer to the DP definition is added to the list of DP definitions. Now it is easy to find the DP definition, which makes it necessary to supervise the incarnation number, when the breakpoint occurs. Eventually the data breakpoints have to be set. When the interesting incarnation is leaved, the breakpoints have to be cleared eventually. This depends on the rest of the data predicates and data traces. The setting and clearing of breakpoints has to be registered in the data breakpoint counters. Breakpoints must not be cleared if the according counter is not equal to zero. If the counter is zero not only the data breakpoints but the pointers to the DP definition in the hash table have to be removed.

2.2.7.3 Exchange Predicates

Supervision of exchange predicates is done by monitor hook procedures. The hook which has to be used is determined by the kind of operation (send, receive, request, release) of the XP-definition. The appropriate procedure is hooked in. Then an entry in the XP-list of the hook handler routine is made, which is a pointer to the XP-definition. Via this list you get immediately the XP-definition, which has to be analyzed. If this list consists of at least one element when the hook handler is called, then the ANALYZER task is waked up. This task analyzes the XP-predicates and does the necessary action, if any. If a XP has to be deleted the pointer has to be dequeued from the list. The dummy procedure has to be hooked in only when all lists of the hook table element are empty.

2.2.7.4 Implementation of Crosstrigger Supervision

Cross triggers (CT) in the SW monitor are done by sending NX/2 broadcast messages with a length of four bytes. The message consists of a unique CT number, which has to be greater or equal to one. To recognize a crosstrigger the interrupt driven receiver routine (hrecv) of the NX/2 is used. When the analyzer task starts working for the first time (after start of the MMK) hrecv (CT_MSG,&ct_in,sizeof(int),ct_handler) is called. CT_MSG is the message type for crosstriggers. “ct_in” is a variable which contains the number of the received crosstrigger.

“ct_handler” is the handler which has to be executed when a crosstrigger message arrives. The handler writes the crosstrigger number to ct_in and wakes up the ANALYZER task (restask(..)). This task takes over the analysis of the crosstrigger. It searches through the list of CT definitions in the current OR block for a CT definition with the gotten number. If it is successful the appropriate action is executed. After handling this crosstrigger ct_in is set to zero and hrecv(...) is called again. If there was another crosstrigger while handling the first, the crosstrigger handler tries to wake up the ANALYZER task, which is still running. Therefore the ANALYZER task has to test the value of ct_in before it suspends itself. If ct_in is not equal to zero the loop has to be executed once more.

2.2.8 Implementation of the Actions

When a predicate is fulfilled or a crosstrigger has arrived the associated actions have to be done. Hereby it is possible that one event fulfills more than one predicate at a time. In this case several actions have to be done. To accomplish this the actions of the fulfilled predicates are gathered in a structure which is of the same type as the action structure associated with the predicate definition. Then all actions are executed together.

2.2.8.1 Block switch Action

This action finishes the supervision of an OR-block and switches to the following block. In the definition of the action there is a parameter, which stands for the relative distance to the following block within the list of OR-blocks. If the parameter is “0" then supervision remains within this block. This has the same effect as if there was no block switch action. ”-1" stops supervision. To activate or deactivate the supervision of an OR-block there are two S0-functions: set_p_watch(pointer to OR-block) and reset_p_watch() respectively. If the OR-block, which should be supervised, is not existent, then an asynchronous error message is sent to the host (error in predicate structure) and supervision is finished. A difficult and not completely solved problem appears, if several predicates demand different blocks to switch to. In this case one predicate is selected and the associated block switch is done. The new block is searched for the predicate whose block switch action was not done. If it is found the block switch is done now. This is not a correct solution!

2.2.8.2 Break Action

The break action is used to stop user tasks. Break uses the MMK system call “__stop_task()” to switch tasks to a stopped state. The stopped state is a combination of the original state and a “stopped” bit, except for the state running, which is switched to “stopped&ready”. The tasks, which should be stopped, are specified with the help of the S1 command “break” before supervision starts . This command is translated to a list of addresses of task token variables. The break action runs through this list and stops theese tasks. If the list is empty, all user tasks are stopped. Stopping is only done for the tasks which are on the processing node where the action is initiated. All other nodes have to be informed by a crosstrigger. This is not included in the break action. The host is informed of the break action by an asynchronous message.

2.2.8.3 Trace-Stop Action

The trace-stop action stops all prior defined traces. There are three possible ways to do this: pre-mode, center-mode and post-mode. In pre-mode the trace buffer contains the trace up to the time, when the trace stop action is done. The trace stop action uses the S0 function “reset_t_watch” to stop all traces. Then the host is informed by an asynchronous message. In center mode tracing is continued for half as many tracing events as the trace buffer is long. Then reset_t_watch is called and the host is informed. The third possibility is the post-mode. In this mode the traces are continued for one full length of the trace buffer. In center- and post-mode it is possible that the tracebuffer is read before tracing is complete. This can happen if events appear rarely. In this case the trace buffer contains probably data at the beginning of the buffer, which was traced before the trace-stop action (post mode), or there is more data before the trace-stop action than afterwards (center mode). Therefore not only the current time but also the index of the next trace buffer entry has to be noted down, when the trace stop action is executed. Using the S1 command “trace-stop-position”, you can get index and time of the latest trace buffer entry before you read the trace buffer. Before sending this command you have to use the command “clear” to disable further tracing. Otherwise new trace data may be added between the commands “trace-stop-position” and reading of the trace buffer. This would invalidate the position of the last entry.

2.2.8.4 Crosstrigger Action

The crosstrigger action sends a NX/2 broadcast message to all nodes. The message consists of the crosstrigger number, which is defined in the definition of the action.

2.2.8.5 Event-Count Action

This action is used for incrementing event counters. The number which was given in definition of the action must be the number of an event counter. The counter is looked for in the list of counter definitions. If it is found, the counter is incremented and the current time is noted as the latest update time.

2.2.9 Implementation of the Single Step Facility

Code addresses with the breakpoint type “single step” reflect the beginning of a high level language statement. To do single stepping in units of the high level language all code addresses, which have the breakpoint type “single step” in the hash table, are replaced by the “int3" breakpoint opcode. This is necessary because generally you can not predict the program flow (conditional jumps, jump tables). Then the whole hash table is looked up to set the breakpoints. If the breakpoint occurs during program execution you have to check the task which is running. If it is the task which was specified in the S1 step command it is stopped. Otherwise execution is continued. The user has the choice between two options: statement step and procedure step respectively. If he chooses statement step, execution is stopped after every breakpoint even if there was a function call (except library calls). The procedure step option only steps within the current function and executes called functions completely. After a single step the stepped task is stopped and the breakpoints are cleared, if they are not used for other purposes. Afterwards the reply to the step command is sent to the host.

2.2.10 Implementation of Trace Supervision

2.2.10.1 Execution Trace

Like single stepping execution traces are accomplished by single step breakpoints. When an interrupt happens the list of execution trace definitions is searched for a matching active definition and if found, the code address is traced.

2.2.10.2 Data Trace

To trace data all code addresses, where the instruction does a memory access with the right type and access direction (reading or writing), have to be substituted by breakpoint interrupts. For the contents of variables are only changed by write accesses, only this type of access is traced. The associated data breakpoint counter has to be incremented. If local variables are traced the current procedure incarnation has to match a given one. This has also to be supervised. To do supervision of incarnations there is a list in the hash table elements, which contains pointers to data trace definitions. So managing data traces is similar to data predicate supervision. When a data breakpoint is reached, the list of data trace definitions has to be searched for a matching definition, eventually followed by tracing the data.

2.2.10.3 Exchange Trace

Administration of exchange traces is done in the same way as for exchange predicates with the help of monitor hooks. The monitor hook table uses a list of pointers to exchange trace definitions, which can be fulfilled by this hook. In contradiction to exchange predicates there is no necessity to wake up the ANALYZER task, but the handler does the analysis of the actual parameters. This is possible because there are no actions which influences the monitored task (see monitor hooks).

2.2.10.4 Performance Traces

Performance Traces are handled like exchange traces.

2.2.11 Implementation of Counters

Event counters are incremented by conditional actions. All other counters are implemented with the help of monitor hooks. Administration is similar to performance or exchange traces. Waiting queue counters have to be initialized with the current length of the queue when started.

2.2.12 The Disassembler

The disassembler is used to track down instructions in the user program, which do memory access. The address, data type and access direction of theese instructions is put in the hash table. The disassembler gets the start and end addresses of all procedures. The memory between these addresses is disassembled with the help of tables, which contain for every code the information about the access direction and instruction length. There are two tables. One table for single byte instructions the other one for two byte instructions. Before the instruction is analysed the prefix has to be handled. If there is an operand size prefix, then the following instruction uses 16 bit operands. Without this prefix the operands have 32 bits. The first byte of the instructions determines whether it is a one or two byte instruction. If it is a two byte instruction the opcode can be found in the second byte. The opcode indexes an entry in the table. This entry contains the length of the instruction for the 16 bit and the 32 bit version in the fields 1_16 and 1_32 without the length of the address mode specifier part of the opcode. For some instructions it is possible to determine whether they do a memory access by looking only at the opcode. For the other instructions the address mode specifier part must be analysed. The entries for instructions which do memory access contain the fields yp and for data type and direction of the memory access. There are only two data types: “byte” and “long”. “word” is type long with operand size prefix. If a memory access is discovered, the relevant data is put in the hash table with “ht_put”. The privileged instructions of the 80386 are not in the opcode tables because they must not appear in user programs.

Appendix A: Description and Syntax of the Interface Layer S1

1 Predicate Action Command:

This command is used to enter and to start predicate monitoring. A predicate and a list of actions form a base unit for monitoring. A group of these units is composed into a block, in which all predicates contained are ORed. Such a block is terminated by entry of the block separation character ">". After this, a new block can be entered. Multiple blocks form a list of "OR blocks". The elements of a list can be connected sequentially-AND by usage of the block switch action (c.f. actions).

SYNTAX:

```
'Z'
    {
        {
            {EP | DP | XP | CT}              /* Predicate specification */

            'A'
            {
                'S' hex32
            |
                'C' hex8
            |
                'B'
            |
                'T' {'V'|'M'|'N'}
            |
                'P' hex8
            }+
            'A'                              /* Action specification */

        }+

    ['> ']                                   /* Separation of 'OR blocks' */
    }+
EOC
```

RESPONSE : '#'*errorcode*

Predicate specification:

Execution predicate

```
EP ::= 'E'
    hex32                       /* Address :  Offset */
    ['#' hex8]                  /* Incarnations */
    ['T' hex32]                 /* Address of task token variable */
    'E'
```

Data Predicate

```
DP ::= 'D'
    {'B'|'W'|'L'}               /* Data type */
    hex32                       /* Address or procedure start address (BCA) */
    {'L'|'G'}                   /* Local-Global */
    [
        ['#' hex8]              /* Incarnation */
        '.' hex32               /* Offset */
        hex32                   /* Procedure end address */
    ]
    {'V' hex | 'R' | 'W'}       /* Comparison with value, read or write access */
    'T'hex32                    /* Address of task token variable */
    'D'
```

Exchange Predicate

```
XP ::= 'X'
    ['T' hex32]                 /* Address of task token variable */
    {
        {''|''}                 /* send, receive */
        ['M' hex32]             /* Address of mailbox variable */
        ['N' hex32]             /* Message pointer */
    |
        {'+'|'-'}               /* release, request */
        ['S' hex32]             /* Address of semaphor variable */
        ['U' hex8]              /* Amount of units */
    }
    'X'
```

Cross trigger

```
CT ::= 'C'
    hex8                        /* No. of logical line to be monitored for CT arrival */
    'C'
```

Action specification:

'S' : Block switch action. Switch to monitoring of another 'OR block'.
(Sequence : e.g. 1=next block, 2=next plus one block ,...)
-1 = clear monitoring of current block !

'C' : Cross trigger action. Issue cross trigger signal on specified logical line

'B' : Break action. Step tasks specified in break list (on own node), all tasks if list is empty (c.f. break command). If the first token address in the break list is 0, only the fulfillment of the predicate will be reported and no task will be stopped. The character '!' (MSIG_BREAKPOINT) will be used as message. If the action occurs during the execution of a step command (see below), the character '$' (MSIG_EXEC_FAILED) will be sent.

'T' : Trace stop action depending on trace mode (pre/center/post 'V','M','N') :
'V' : Stop trace
'M' : Trace continues for half a buffer length
'N' : Trace continues for full a buffer length
After execution of the trace, the trace supervision will be cleared and the trace end will be reported asynchronously to the host machine. For that purpose the same characters as for break action will be used.

'P' : Event count action (specified counter will be incremented by 1)

2 Break command

The break command is used for entry of tasks to be stopped during a break action.

During action 'break' the tasks specified in the stop list will be stopped. If the list is empty, all tasks but the monitor tasks, server tasks, idle tasks will be stopped.

SYNTAX:

```
'B'
    {
        hex32                                  /* Address of task token variable */
    }*
EOC
```

RESPONSE : '#'*errorcode*

3 Trace command

The trace command is used for definition and (if needed) for starting of traces.

SYNTAX:

```
'T'
    {ET | DT | XT | 1| ...| 13}          /* Trace type */
    {'S'|'I'}                            /* Start mode */
EOC
```

RESPONSE : '#'*errorcode* [*hex8*] /* Trace no .*/

Start mode :

S : after entry of a predicate specifion (with predicate action command)

I : immediately after entry of trace definition

Trace type:

Execution trace:

```
ET ::= 'E'
    ['T'hex32]                           /* Address of task token variable */
    'E'
```

Recorded will be :

- Trace no.
- Task token
- 0
- IP
- Time

Data trace:

```
DT ::= 'D'
    {'B'|'W'|'L'}                        /* Data type */
    hex32                                /* Address or BCA */
    {'L'|'G'}                            /* Local/global */
    [
        ['#'hex8]                        /* Incarnation */
        '.'hex32                         /* Offset */
        hex32                            /* BCA End */
    ]
    'T'hex32                             /* Address of task token variable */
    'D'
```

Recorded will be :

- Trace no.
- 0
- 0
- Variable content (value)
- Time

Exchange trace:

```
XT ::= 'X'
    {
        'M'
        hex32               /* Address of mailbox token variable */
        {'>'|'<'}           /* send, receive */
        {'T'|'N'}           /* Recording of task or message pointer */
                            /* this choice has become obsolete, because
                            both values will be recorded anyway, but for
                            compatibility reasons it still has to be stated. */
    |
        'S'
        hex32               /* Address of semaphor token variable */
        {'+'|'-'}           /* release, request */
        {'T'|'U'}           /* Recording of task or semaphor units */
                            /* this choice has become obsolete, see above.
                            ! */
    |
        'T'
        hex32               /* Address of task token variable */
        {'>'|'<'|'+'|'-'}   /* send, receive, release, request */
    }
    'X'
```

Case 'M': Recorded will be :

- Trace no.
- Task token
- 0
- Message pointer
- Time

Case 'S': Recorded will be :

- Trace no.
- Task token
- 0
- Semaphor units
- Time

Case ’T’: Recorded will be :

- Trace no.
- Semaphor token or mailbox token (depending on operation)
- 0
- 0
- Time

Performance traces :

Trace: task begins computation

1 ::= ’R’
['T'*hex32*] /* Address of task token variable, if omitted all tasks on the node will be monitored */
’P’

Recorded will be:

- Trace no.
- Task token
- 0
- 0
- Time

Trace: task stops computation

2 ::= ’S’
['T'*hex32*] /* Address of task token variable, if omitted all tasks on the node will be monitored */
’P’

Recorded will be:

- Trace no.
- Task token
- 0
- 0
- Time

Trace: task requests message from mailbox

3 ::= ’B’
['T'*hex32*] /* Address of task token variable */
['O'*hex32*] /* Address of mailbox token variable */
’P’

Recorded will be :

- Trace no.
- Task token
- Mailbox token
- 0
- Time

Trace: task gets message from mailbox

```
4 ::= 'G'
    ['T'hex32]                /* Address of task token variable */
    ['O'hex32]                /* Address of mailbox token variable */
    'P'
```

Recorded will be :

- Trace no.
- Task token
- Mailbox token
- Length of message
- Time

Trace: task starts sending of message to mailbox

```
5 ::= 'A'
    ['T'hex32]                /* Address of task token variable */
    ['O'hex32]                /* Address of mailbox token variable */
    'P'
```

Recorded will be :

- Trace no.
- Task token
- Mailbox token
- Length of message
- Time

Trace: Task terminates sending of message to mailbox

```
6 ::= 'C'
    ['T'hex32]                /* Address of task token variable */
    ['O'hex32]                /* Address of mailbox token variable */
    'P'
```

Recorded will be :

- Trace no.
- Task token
- Mailbox token

- Length of message
- Time

Trace: Mailbox gets request for message from task

7 ::= ’F’
['T'*hex32*] /* Address of task token variable */
['O'*hex32*] /* Address of mailbox token variable */
’P’

Recorded will be :

- Trace no.
- Task token
- Mailbox token
- 0
- Time

Trace: mailbox sends message to task

8 ::= ’H’
['T'*hex32*] /* Address of task token variable */
['O'*hex32*] /* Address of mailbox token variable */
’P’

Recorded will be :

- Trace no.
- Task token
- Mailbox token
- Lenght of message
- Time

Trace: task begins P operation (req) on semaphore

9 ::= ’I’
['T'*hex32*] /* Address of task token variable */
['O'*hex32*] /* Address of semaphore token variable */
’P’

Recorded will be :

- Trace no.
- Task token
- Semaphor token
- Amount of requested units
- Time

Trace: Task terminates P operation (req) on semaphore

```
10 ::= 'J'
    ['T'hex32]          /* Address of task token variable */
    ['O'hex32]          /* Address of semaphore token variable */
    'P'
```

Recorded will be :

- Trace no.
- Task token
- Semaphor token
- Amount of requested units
- Time

Trace: task begins to wait at semaphore

```
11 ::= 'K'
    ['T'hex32]          /* Address of task token variable */
    ['O'hex32]          /* Address of semaphore token variable */
    'P'
```

Recorded will be :

- Trace no.
- Task token
- Semaphor token
- 0
- Time

Trace: semaphor releases task (no timeout !)

```
12 ::= 'L'
    ['T'hex32]          /* Address of task token variable */
    ['O'hex32]          /* Address of semaphore token variable */
    'P'
```

Recorded will be :

- Trace no.
- Task token
- Semaphor token
- 0
- Time

Trace: semaphor gets value by V operation (rel) of a task

```
13 ::= 'O'
    ['T'hex32]                      /* Address of task token variable */
    ['O'hex32]                      /* Address of semaphore token variable */
    'P'
```

Recorded will be :

- Trace no.
- Task token
- Semaphor token
- Semaphor value
- Time

4 Clear command

SYNTAX:

```
'C'
    {
        'Z'                         /* deactivate predicate action command*/
    |
        'T'
        {hex8}*                     /* clear traces with specified no. */
    }
EOC
```

RESPONSE: '#'*errorcode*

5 Reset trace buffer command

SYNTAX:

```
'R'                                 /* reset trace buffer */
EOC
```

RESPONSE : '#'*errorcode*

6 Read trace buffer

SYNTAX:

```
    'U'                                      /* Output trace buffer */
    EOC
```

Response : '#'*errorcode*

```
        hex32                                /* Trace start time */
        hex32                                /* Time of trace buffer readout */
        {                                    /* Contents of trace buffer */
            hex32                            /* Trace no */
            hex32                            /* global ID */
            hex32                            /* global ID */
            hex32
            hex32                            /* Time */
        }*
```

c.f. trace, recorded will be: ...

7 Trace stop position command

The command is used for output of the position of the element in the tracebuffer, which has been recorded immediately after trace trigger.

If need be, this command can be used immediately before (!) 'read trace buffer'. Before using this command, active traces should be cleared with the clear command. The data of the trace stop position command could otherwise be invalidated by further recordings.

SYNTAX:

```
    'H'
    EOC
```

RESPONSE :

```
        '#' errorcode
        hex32                                /* Index (e.g. 3 : Trace stop action before the
                                             third element in the trace buffer */
        hex32                                /* Time of trace stop action  */
```

8 Set time unit command

This command is used to set the time base for the timer.

SYNTAX:

```
'F'{hex32}                          /* Time base in µs */
EOC
```

RESPONSE: '#' *errorcode*

9 Activate counter command

Activate integrating counter for:

a) Process queue

b) Process queue at a mailbox

c) Process queue at a semaphore

d) Sent message units (NX message length in bytes)

e) Active time of idle task

f) Event count

g) Process queue at remote procedure calls

SYNTAX:

```
'J'
    {
        'T'                         /* Counter type a) */
    |
        'M'[hex32]                  /* Address of mailbox token variable - Counter
                                       type b) */
    |
        'S'[hex32]                  /* Address of semaphore token variable -
                                       Counter type c) */
    |
        'U'[hex32]                  /* Node no. (-1 = Node no. doesn't matter)
                                       - Counter type d) */
    |
        'I'                         /* Idle task - Counter type e) */
    |
        'E'                         /* Event count - Counter type f) */
    |
```

```
        'R'                          /* Counter type g)   */
    }
  EOC
```

RESPONSE: '#' *errorcode* [*hex8*] /* Counter no. */

10 Read counter command

This command is used to read out counter values

SYNTAX:

```
  'K'{hex8}                          /* Counter no. */
  EOC
```

RESPONSE:

```
    '#' errorcode
    [
        hex32                        /* Value of counter */
        hex32                        /* Start time of the counter (will be recorded
                                        with each read) */
        hex32                        /* Readout time of the counter */
    ]
```

11 Reset counter command

Deactivate/reset specified counter. All counters will be deactivated when counter no is '0'

SYNTAX:

```
  'L'{hex8}                          /* Counter no.  */
  EOC
```

RESPONSE: '#' *errorcode*

12 Display command

SYNTAX:

```
'D'
    {
        {
            'M'                          /* Data segment */
        |
            'C'                          /* Code segment */
        }
        {
            'B'                          /* Byte */
        |
            'W'                          /* Word */
        |
            'L'                          /* Long */
        }                                /* data type */
        hex32                            /* Start address of memory area */
        [hex32]                          /* Amount of components */
    }
  EOC
```

RESPONSE:

```
    '#' errorcode
    {hex}*                               /* Contents */
```

13 Modify command

SYNTAX:

```
'M'
    {
        {
            'M'                          /* Data segment */
        |
            'C'                          /* Code Segment */
        }
        {
            'B'                          /* Byte */
        |
            'W'                          /* Word */
        |
            'L'                          /* Long */
        }                                /* data type */
        hex32                            /* Start address of memory area */
```

```
        {hex}+                              /* List with new values*/
    }
  EOC
```

RESPONSE : '#' *errorcode*

14 Symbol command

"symbol" gives information about a task's stack. If the subcommand is 'L', the BCA-list specifies an absolute path from root of the stack to a specific function call. The result is the absolute address of a local variable with the given base pointer offset. For the 'V subcommand only a single BCA is specified; "symbol" gives the absolute address of a local variable in the currently active incarnation. With the third possible subcommand 'S' you get a list of all BCA's on the stack of the task given by its token.

SYNTAX:

```
  'Y' hex32                                 /* Task token */
    {
        'L'
        {hex32 ['#' hex] }+                 /* List of BCA and incarnations */
        '.' hex32                           /* Offset */
    |
        'V'
        hex32                               /* BCA */
        '.' hex32                           /* Offset */
    |
        'S'
    }
    EOC
```

RESPONSE:

```
    '#' errorcode
    {hex32}*                                /* List of BCAs or an absolute address */
```

15 View command

“view” gives a list of all MMK objects of the specified type, selected by object, i.e. tasks, mailboxes, semaphores.

SYNTAX:

```
'V'
    {'T'|'M'|'S'}                /* Object type: task, mailbox, semaphore */
 EOC
```

RESPONSE:

```
    '#' errorcode
    {hex32}*                     /* List of tokens of MMK objects */
```

16 Inspect command

“inspect” gives more specific data about a MMK object, with given type and token. The information about tasks is composed of priority, scheduling state, current execution point and, if the task waits at a mailbox or a semaphore, its token and the message pointer or semaphore units. The data for mailbox objects are total buffer locations, free buffer locations, buffer contents and the queue of waiting tasks. For semaphores the output are the current units and the queue of waiting tasks.

SYNTAX

```
'I'
    {
        'T'hex32                 /* Task token */
        hex32                    /* last BCA on the stack */
    |
        {'M'|'S'}
        hex32                    /* Mailbox/Semaphore token */
    }
 EOC
```

RESPONSE: '#' *errorcode*

```
    {                            /* Task: */
        hex32                    /* Priority of the task */
        hex32                    /* State of the task    */
        hex32                    /* Token of the mailbox/semaphore, if the task
                                    is currently waiting */
```

```
        hex32                       /* Message pointer */
        hex32                       /* Units */
        hex32                       /* IP */
    |                               /* Mailbox: */
        hex32                       /* Total buffer locations */
        hex32                       /* Free buffer locations */
        {hex32}*                    /* Buffer contents */
        {hex32}*                    /* Queue */
    |                               /* Semaphore: */
        hex32                       /* Units */
        {hex32}*                    /* Queue */
    }
```

17 Go command

"go" starts all tasks specified in the specified task list. If it is empty, all get started.

SYNTAX:

```
  'G' {hex32}*                      /* List of task token */
  EOC
```

RESPONSE : ' #' *errorcode*

18 Step command

The step command is used for single step execution of a task given by its token.The step type ('S' or 'P') determines the handling of procedure calls. If the type is 'S', execution continues until the next statement is reached, that forms the beginning of a source code line. The 'P'-mode works the same way, but treats procedure calls as single statements, so that the called procedure is not single stepped.

SYNTAX:

```
    {
        'S' hex32                   /* Task token */
    |
        'P' hex32                   /* Task token */
        hex32                       /* current IP of the task */
        hex32                       /* End address of the procedure */
    }
  EOC
```

RESPONSE: '#' *errorcode*

19 Initdeb command

"initdeb" does the necessary initializations of the SWM by issueing the S0 function "hwinit()". A list of all procedure start, procedure end and statement start addresses of the monitored program has to be supplied. This list is mainly used for the single step command.

SYNTAX:

```
'A'
    {                                   /* Address list of high level language statements */
        {
            'P'                         /* Procedure start */
        |
            'S'                         /* Statement start */
        |
            'E'                         /* Procedure end */
        }
    hex32                               /* Address */
    }+
EOC
```

RESPONSE : '#' *errorcode*

Explanations :

hex8	Hex number 8 bit + 1 blank
hex32	Hex number 32 bit + 1 blank
hex	Hex number 8,16, oder 32 bit + 1 blank
errorcode	Error number as in file errormsgs.h
[...]	optional parameter
{ ... }	1 occurence of the brace's contents
{ ... }+	>=1 occurence of the brace's contents
{ ... }*	>= 0 occurence of the brace's contents

Appendix B: Features of the S0 Interface

FUNCTION:

_ _set_p_watch

Syntax:

```
Errmsg  _ _set_p_watch (p)
pal_elem  * p;
```

Semantics:

_ _*set_p_watch* sets BPs for predicates of the block currently under supervision

Input:

p : Pointer to the block to be supervised

Output:

none

FUNCTION:

_ _reset_p_watch

Syntax:

```
void _ _reset_p_watch ()
```

Semantics:

_ _*reset_p_watch* clears BPs for predicates in currently inspected block

Input:

none

Output:

none

FUNCTION:

__set_1t_watch

Syntax:

```
Errmsg  __set_1t_watch (tptr,type)
char  * tptr, type;
```

Semantics:

__set_1t_watch sets BPs for a trace

Input:

tptr :	Pointer to trace definition
type :	Trace type 'E','D','X','P'

Output:

Error message

FUNCTION:

__reset_1t_watch

Syntax:

```
void __reset_1t_watch (tptr,type)
char  * tptr, type;
```

Semantics:

__reset_1t_watch clears BPs for a trace

Input:

tptr :	Pointer to trace definition
type :	Trace type

Output:

none

FUNCTION:

_ _set_cntr_watch

Syntax:

```
Errmsg  _ _set_cntr_watch (cntr)
cntr_elem  * cntr;
```

Semantics:

_ _*set_cntr_watch* sets supervision for a counter

Input:

cntr: Pointer to counter definition

Output:

Error message

FUNCTION:

_ _reset_cntr_watch

Syntax:

```
void  _ _reset_cntr_watch (cntr)
cntr_elem  * cntr;
```

Semantics:

_ _*reset_cntr_watch* clears supervision for a counter

Input:

cntr : Pointer to counter definition

Output:

Error message

FUNCTION:

_ _read_cntr

Syntax:

```
void  _ _read_cntr (ptr,pval,pst,prt)
cntr_elem  * ptr;
long  * pval,* pst,* prt;
```

Semantics:

_ _*read_cntr* read the counter pointed to

Input:

ptr:	Pointer to a counter
pval :	Pointer to variable to contain the current counter value
pst :	Pointer to variable to contain the starting time of the counter
prt :	Pointer to variable to contain the readout time of the counter

Output:

none

FUNCTION:

__read_timer

Syntax:

```
long  _ _read_timer ();
```

Semantics:

_ _*read_timer* reads the clock used in the monitor

Input:

none

Output:

current time

FUNCTION:

__set_time_unit

Syntax:

```
void __set_time_unit (tu);
long tu;
```

Semantics:

__set_time_unit is used to enter a new time base for the clock used in the monitor

Input:

New time base in microseconds

Output:

none

FUNCTION:

__ttpos

Syntax:

```
Errmsg  __ttpos (vor_el_nr,zeit);
int   * vor_el_nr;
long  * zeit;
```

Semantics:

__ttpos queries at which element in the tracebuffer and when the trace stop action occurred

Input:

vor_el_nr :	Pointer to variable to contain the index of the trace buffer element
zeit :	Pointer to variable to contain the time mark

Output:

Error message (if no trace stop action occurred)

FUNCTION:

_ _res_trace_buffer

Syntax:

void _ _res_trace_buffer ();

Semantics:

_ _*res_trace_buffer* is used to clear the trace buffer

Input:

none

Output:

none

FUNCTION:

_ _tbcount

Syntax:

int _ _tbcount ();

Semantics:

_ _*tbcount* returns amount of elements in the trace buffer

Input:

none

Output:

Amount of trace buffer entries

FUNCTION:

__tbinhalt

Syntax:

void __tbinhalt (tbval)
tb_elem * tbval;

Semantics:

__*tbinhalt* returns the content of the next (FIFO) element in the trace buffer

Input:

tbval: Pointer to a variable to contain the five components of the trace buffer element

Output:

none

FUNCTION:

__inithw

Syntax:

void __inithw ();

Semantics:

__*inithw* does initializations in the monitor

Input:

none

Output:

none

Appendix C: Monitoring hooks within the MMK

The following text describes the monitoring hooks and the modules within the MMK source code tree where the hooks are located. In addition every hook description is augmented by a list of predicates, traces and counters, which make use of this hook. Path names are relative to the directory MMK/CUBE/. For a description of predicates, traces and counters see appendix A.

1. ———————————————————————————-

HOOK FUNCTION CALL:

```
(* _mh_schedu) (_oldrunptr);
task_descr *  _oldrunptr;
```

MODULE:

MANAG/STATE/sch_st.c

FUNCTION:

_sched ()

USED FOR:

Traces: <1>, <2>
Counter: e)

2. ———————————————————————————-

HOOK FUNCTION CALL:

```
(* _mh_snd_beg) (mbox_id,msg_ln,msg_addr);
MBOX_ID mbox_id;
int msg_ln;
char * msg_addr;
```

MODULE:

USER/MBOX/snd_mb.c

FUNCTION:

sndmsg ()

(before disable(), because of task switch on XP)

USED FOR:

Traces: <5>, XT ’T’, XT ’M’
Predicates: XP ’>’

3. ——————————————————————-

HOOK FUNCTION CALL:

(* _mh_snd_end1) (mbox_id,msg_ln);
(* _mh_snd_end2) (mbox_id,msg_ln);
MBOX_ID mbox_id;
int msg_ln;

MODULE:

USER/MBOX/snd_mb.c

FUNCTION:

sndmsg ()

USED FOR:

Traces: <6>

4. ——————————————————————-

HOOK FUNCTION CALL:

(* _mh_rec_beg) (orig_mbox_id);
MBOX_ID orig_mbox_id;

MODULE:

USER/MBOX/rec_mb.c

FUNCTION:

recmsg ()

USED FOR:

Traces: <3> , <7>

5. ——————————————————————————————-

HOOK FUNCTION CALL:

(* _mh_rec_end1) (orig_mbox_id,msg_ln,msg_addr);
(* _mh_rec_end2) (orig_mbox_id,msg_ln,msg_addr);
MBOX_ID orig_mbox_id;
int msg_ln;
char *msg_addr;

MODULE:

USER/MBOX/rec_mb.c

FUNCTION:

recmsg ()

USED FOR:

Traces: <4> , <8> , XT 'T' , XT 'M'
Predicates: XP '<'

6. ——————————————————————————————-

HOOK FUNCTION CALL:

(* _mh_snd_1) (mbox_id);
(* _mh_snd_2) (mbox_id);
MBOX_ID mbox_id;

MODULE:

USER/MBOX/snd_mb.c

FUNCTION:

sndmsg ()

USED FOR:

Counter: b)

7. ———————————————————————————-

HOOK FUNCTION CALL:

(* _mh_rec_1) (orig_mbox_id);
(* _mh_rec_2) (orig_mbox_id);
MBOX_ID orig_mbox_id;

MODULE:

USER/MBOX/rec_mb.c

FUNCTION:

recmsg ()

USED FOR:

Counter: b)

8. ———————————————————————————-

HOOK FUNCTION CALL:

(* _mh_req_beg) (orig_id,units);
SEMA_ID orig_id;
unsigned units;

MODULE:

USER/SEMA/req_se.c

FUNCTION:

reqsema ()
(before disable(), because of task switch to XP)

USED FOR:

Traces: <9> , XT 'T' , XT 'S'
Predicates: XP '+'

9. ———————————————————————-

HOOK FUNCTION CALL:

(* _mh_req_end1) (orig_id,units);
(* _mh_req_end2) (orig_id,units);
SEMA_ID orig_id;
unsigned units;

MODULE:

USER/SEMA/req_se.c

FUNCTION:

reqsema ()

USED FOR:

Traces: <10>

10. ———————————————————————-

HOOK FUNCTION CALL:

(* _mh_rel_beg) (id,units);
SEMA_ID id;
unsigned units;

MODULE:

USER/SEMA/rel_se.c

FUNCTION:

relsema ()
(before disable(), because of task switch to XP)

USED FOR:

Traces: XT ’T’ , XT ’S’
Predicates: XP ’-’

11. ——————————————————————————-

HOOK FUNCTION CALL:

(* _mh_rel_end) (id,s_ptr->value);
SEMA_ID orig_id;
unsigned s_ptr->value;

MODULE:

USER/SEMA/rel_se.c

FUNCTION:

relsema ()

USED FOR:

Traces: <13>

12. ——————————————————————————-

HOOK FUNCTION CALL:

(* _mh_req_se1) (orig_id);
SEMA_ID orig_id;

MODULE:

USER/SEMA/req_se.c

FUNCTION:

reqsema ()

USED FOR:

Traces: <11>
Counter: c)

13. ——————————————————————————-

HOOK FUNCTION CALL:

(* _mh_tst_se1) (t_ptr,s_ptr);
task_descr * t_ptr;
sema_descr * s_ptr;

MODULE:

MANAG/QUEUE/del_qu.c

FUNCTION:

_tst_se ()

USED FOR:

Traces: <12>
Counter: c)

14. ———————————————————————-

HOOK FUNCTION CALL:

(* _mh_del_qu1) ((mbox_descr *) o_ptr);
obj_top * o_ptr;

MODULE:

MANAG/QUEUE/del_qu.c

FUNCTION:

_del_qu ()

USED FOR:

Counter: b)

15. ———————————————————————-

HOOK FUNCTION CALL:

(* _mh_ins_rdy) (t_ptr);
task_descr * t_ptr;

MODULE:

MANAG/STATE/ins_st.c

FUNCTION:

_ins_rd ()

USED FOR:

Counter: a)

16. ———————————————————————-

HOOK FUNCTION CALL:

```
(* _mh_del_rdy) (t_ptr);
task_descr  *t_ptr;
```

MODULE:

MANAG/STATE/del_st.c

FUNCTION:

_del_rd ()

USED FOR:

Counter: a)

17. ———————————————————————-

HOOK FUNCTION CALL:

```
(* _mh_rmk_snd) (node,length);
int  node;
int  length;
```

MODULE:

GLOBAL/NX/nx.c

FUNCTION:

_rmk_send ()

USED FOR:

Counter: d)

18. ———————————————————————-

HOOK FUNCTION CALL:

```
(* _mh_rem_snd) (mboxid);
ID mboxid;
```

MODULE:

GLOBAL/REMOTE/request.c

FUNCTION:

_remote_send ()

USED FOR:

Counter: g)

19. --

HOOK FUNCTION CALL:

(* _mh_rem_act) (objid);
ID objid;

MODULE:

GLOBAL/REMOTE/request.c

FUNCTION:

_remote_action ()

USED FOR:

Counter: g)

20. --

HOOK FUNCTION CALL:

(* _mh_sp_rem_act) (objid);
ID objid;

MODULE:

GLOBAL/REMOTE/request.c

FUNCTION:

_special_remote_action ()

USED FOR:

Counter: g)

21. ——————————————————————————-

HOOK FUNCTION CALL:

(* _mh_rem_cre_ta) ();

MODULE:

GLOBAL/REMOTE/request.c

FUNCTION:

_rem_cre_ta ()

USED FOR:

Counter: g)

22. ——————————————————————————-

HOOK FUNCTION CALL:

(* _mh_srv_ans) ();
(* _mh_srv_rec_ans) ();

MODULE:

GLOBAL/REMOTE/server.c

FUNCTION:

_rem_handler ()

USED FOR:

Counter: g)

23. ——————————————————————————-

HOOK FUNCTION CALL:

(* _mh_mon_start) ();

MODULE:

MANAG/SYSTEM/run_sy.c

FUNCTION:

_runmmk ()

(before creation of the idle task)

USED FOR:

start of the monitor, when the MMK is started

(task_descr *) _runptr and the ID of the running task, respectively is used in the handlers of the hooks with the numbers mentioned below:

1, 2, 3, 4, 5, 8, 9, 10, 11

Part 6:

Source File Structure And Program Generation

Olav Hansen

The following chapters provide a quick reference on the structure and contents of the TOPSYS source-code directories. In addition, a short description how to generate the various libraries and programs of the TOPSYS environment is given.

1 Structure of the Directory Tree

The "TOPSYS" directory tree is divided up into the subtrees "CUBE", "HOST" and "SRM". All of these subtrees contain the sources for that parts of the tools that have to be executed on the according machines. The "HOST" directory tree is subdivided into the trees "GRAPHIK", "TOOLS" and "UTOL".

The "TOOLS" tree contains the sources for the execution control of the different tools. The "VISTOP" subtree contains the sources for the visualization system "VISTOP", the "PATOP" subtree for the performance analyzing tool "PATOP" and the "DETOP" subtree for the debugging tool "DETOP". The "TOPCOM" directory contains parts that are used by the execution control of all tools. "S5_EQUI" contains software parts needed to produce simulated versions of the tools.

The "UTOL" subdirectory of the "HOST" directory contains the software that is used by the execution control components of all tools to interface the monitors running on the iPSC/2, this software does the transformation between the machine oriented sight required by the monitors and the high level language sight of the interface to the execution control. This software is used by all tools. The "UTOL" directory is subdivided into the "HLTL" and "UDTL" directory trees. The "UDTL" directory tree contains the software parts that transform between the monitor interface and an interface that is symbolic but not special to a certain high level language. The transformation to the special high level language is done by the software contained in the "HLTL" tree.

The "GRAPHIK" subdirectory of the "HOST" directory contains the software responsible for the graphical output. The "MENU", "MESSAGE", "BARGRAPH", "HISTO", "INOUT", "GENOUT" directories contain software that offers an object oriented interface to the graphical object classes menu, message window, bargraph window, curve diagram window, input/output window and pure output window. The execution control components of the tools access these graphical objects through C++ interfaces. The INCLUDE directory contains common header files of the graphical objects. The "WIDGET" directory contains sources for X-Toolkit based Widgets that are used as basis for the graphical object classes mentioned before.

The SRM directory contains the sources for a program to extract symbolic information from an object file that can be run on the iPSC/2. The information is written to a file in a format that is known by the tools.

The CUBE directory tree contains the iPSC parts of TOPSYS. In the HW subdirectory there are specifications for the hardware monitor. The SW subtree contains all software for the use of the software monitor and the hardware monitor. The UTL directory contains that parts of the cube software that are used with both monitors. The C.HWM directory contains the control software for the hardware monitor. The C.HWMANALYZER directory contains the software that reacts on events asserted by the hardware monitor. The SWM directory contains the software monitor. The ANA directory contains the software that reacts on events asserted by the software monitor.

2 Making the Tools

A “make all” in the “HOST” directory produces actual versions of all tools separately and combined. As first step in this compilation process libraries with the names Utol.a, Widgets.a, Menu.a, Message.a, Bargraph.a, Histo.a, InOut.a, GenOut.a, S5_Equi.a, Topcom.a, Detop.a, Patop.a and Vistop.a are updated or created if they didn‘t exist before. A “make all” in one of the directories “HOST/UTOL”, “HOST/GRAPHIK/WIDGETS”, “HOST/GRAPHIK/MENU”, “HOST/GRAPHIK/MESSAGE”, “HOST/GRAPHIK/BARGRAPH”, “HOST/GRAPHIK/HISTO”, “HOST/GRAPHIK/INOUT” or “HOST/GRAPHIK/GENOUT” updates only the according library. A “make all” in “HOST/TOOLS/DETOP”, “HOST/TOOLS/PATOP” or “HOST/TOOLS/VISTOP” produces the according libraries and executable objects. A “make simulation” in these directories produces the according simulated versions. When making one of the simulated versions also the library S5_Equi.a is updated with the sources in “HOST/TOOLS/S5_EQUI”. Always when making one of the tools the library Topcom.a is updated with the sources in “HOST/TOOLS/TOPCOM”. A “make all” in “HOST/TOOLS” produces a new version of the combined tool object. The compiled objects are put into the directory “$(ROOT)/bin”. “ROOT” is a variable that has to be defined in the makefile of the directory where the make is executed. Libraries are put into the “$(ROOT)/lib” directory and header files are put into the “$(ROOT)/include” directory. All dependencies between subtrees are resolved through the “$(ROOT)/lib” and “$(ROOT)/include” directories. There are only source files within the “HOST” directory tree. When making, the age of every source file is compared with the age of the object contained in the according library. Only if the source file is newer than the object in the library it is complied, then the object is copied into the library and removed from the source directory. Using NFS this allows us to work in an efficient way with object versions for different machines while having only one source code version.

The SRM and cube parts of TOPSYS have to be compiled on the SRM. To update the libraries for software monitor or hardware monitor you have to execute a “make swmlib” or “make hwmlib” in the directory “CUBE/SW”. The directory where the library to update is searched is defined by the variable TARGETLIB in “CUBE/SW/makefile”.

References

[Art89] Y. Artsy; R. Finkel: Designing a Process Migration Facility: The Charlotte Experience; IEEE Computer, Sept.1989, p. 47-56

[Bai87] Baiardi, F.; Pelagatti, S.; Vanneschi, M.: Processor Managment in Highly Concurrent Systems, Esprit 415, Sep. 1987

[BeL90] T. Bemmerl; T. Ludwig: MMK - A Distributed Operating System Kernel with Integrated Dynamic Loadbalancing; H. Burkhart; Lecture Notes in Computer Science, Vol 457; Berlin 1987; p. 744 - 755

[Bem87] Bemmerl, T.; Schöder, G.: A Portable Realtime Multitasking Kernel for Embedded Microprocessor Systems; Microprocessing and Microprogramming, The Euromicro Journal; Vol:21; Amsterdam, Aug. 1987; p. 181 - 188

[Bem88] Bemmerl, T.: An Integrated and Portable Tool Environment for Parallel Computers, IEEE Int. Conference on Parallel Processing, St. Charles, August 1988

[Bod85] Bode, A.; Fritsch, G.; Henning, W.; Volkert, J.: High Performance Multiprocessor Systems for Numerical Simulation, IEEE Parallel Processor Conference, St. Charles, 1985

[Bod91a] Bode, A. (ed.): Distributed Memory Computing, Proceedings of EDMCC2, Springer LNCS, 487, 1991

[Bod91b] Bode, A.: Load Balancing in Distributed Memory Multiprocessors, in: Monaco, Negrini (eds.): Advanced Computer Technology, Reliable Systems and Applications, Proceedings COMP EURO 91, 131, 1991

[BHL90] T. Bemmerl; O.Hansen: T. Ludwig: PATOP for Performance Tuning of Parallel Programs; H. Burkhart; Lecture Notes in Computer Science, Vol 457; Berlin 1987; p. 840 - 851

[BLT90] T. Bemmerl; R. Lindhof; T. Treml: The Distributed Monitor System of TOPSYS; H. Burkhart; Lecture Notes in Computer Science, Vol 457; Berlin 1987; p. 756 - 765

[Bur88] H. Burkhart; R. Millen: Techniken und Werkzeuge der Programmbeobachtung am Beispiel eines Modula-2 Monitorsystems; Informatik Forschung und Entwicklung, 3; 1988; p. 6 - 21

[Cor89] J. McCormack; P. Asente; R. R. Swick: X Toolkit Intrinsics - C Language Interface; X Window Systems; X Version 11; Release 4; Cambridge Massachusetts; 1990

[Fri88] G. Fritsch; T. Ludwig; J. Volkert: Many Particle Problems on Distributed Shared Memory Systems; Proceedings of the CONPAR 88; Manchester, UK; Vol A

[Gil88] W. K. Giloi: The Suprenum Architecture: CONPAR 88; Sept. 88; Manchester, UK

[Gua89] V. A. Guarna; d. Gannon; D. Jablonowski; A. d. Malony; Y. Gaur: Faust: An Integrated Environment for Parallel Programming; IEEE Software; July 1989; p. 20 - 26

[Hof87] R. Hofmann; R. Klar; N. Luttenberger; B. Mohr: ZÄHLMONITOR 4: Ein Monitorsystem für das Hardware- und Hybrid-Monitoring von Multiprozessor- und Multicomputer-Systemen; Messung und Modellierung von Rechensystemen; Erlangen; IFB; 1987; p. 79 - 99

[Jes89] C. Jesshope: Large Scale Concurrent Computations: Systems and Software; Univerity of Southhampton; Techn. Report; 1989

[Kle88] J. M. Kleinhans; G. Sigl; F. M. Johannes: GORDIAN: A New Global Optimazation/ Rectangle Dissection Method for Cell Placement; IEEE Int. Conference on Computer-Aided Design ICCAD-88

[Leh89] T. Lehr; Z. Segall; D. F. Vrsalowic; E. Caplan; A. L. Chung; C. E. Fineman: Visualizing Performance Debugging: IEEE Computer; Oct. 1989; p. 38 - 51

[LI89] Li, K.; Schaefer, R.: A Hypercube Shared Virtual Memory System, Proceedings of the 1989 International Conference on Parallel Processing, Vol.I, pp. 125-132, The Pennsylvannia State University Press, University Park

[Mae86] E. Maehle; K. Moritzen; K. Wirl: Fault-Tolerant Hardware Configuration Management on the Multiprocessor System DIRMU 25; CONPAR 86; Lecture Notes on Computer Science; 1986; p. 190 - 197

[ORei90] O’Reilly & Associates, Inc; The X Window System; Vol. 0 - 7; Sebastopol, CA, 1990

[Pap87] Papandopoulus: The New Dataflow Architecture Beeing Built at MIT; Proceedings of MIT-ZTI-Symp. on “Very High Parallel Architectures”; Nov. 1987

[Ros88] Rost, J.; Maehle, E.: Implementation of a Parallel Branch-and-Bound Algorithm for the Travelling Salesman Problem, Proceedings of the CONPAR 1988, Manchester, UK, Vol.: A

[The86] Theimer, M.M.: Preemtable Remote Execution Facilities for Loosely-Coupled Distributed Systems, Stanford University, Report No. STAN-CS-86-1128 (also CSL-86-302), Stanford, 1986