C.1 RELEASABLE
DOC-CR-SP-90-001

Harnessing Knowledge with Technology

CompEngServ

PERFORMANCE ANALYSIS
OF
THE FTDCS SIMULATOR & OPERATING SYSTEM

March 6, 1990

IC

By:

G. Ram, D. Bowen, B.A. Bowen
CompEngServ Ltd.
Suite 600, 265 Carling Avenue
Ottawa, Ontario
Canada K1S 2E1
(613) 563-1920

LKC
QA
76.9
.F38
R36
1990
c.2

.1 RELEASABLE
DOC-CR-SP-90-001

Harnessing Knowledge with Technology

CompEngServ

Industry Canada
Library - Queen

SEP 2 3 2013

Industrie Canada
Bibliothèque - Queen

PERFORMANCE ANALYSIS

OF

THE FTDCS SIMULATOR & OPERATING SYSTEM

March 6, 1990

COMMUNICATIONS CANADA
CRC

MAR 27 1990

LIBRARY -- BIBLIOTHÈQUE

By:

G. Ram, D. Bowen, B.A. Bowen
CompEngServ Ltd.
Suite 600, 265 Carling Avenue
Ottawa, Ontario
Canada K1S 2E1
(613) 563-1920

Government of Canada Gouvernement du Canada

Department of Communications

DOC CONTRACTOR REPORT **DOC-CR-SP-90-001**

DEPARTMENT OF COMMUNICATIONS - OTTAWA - CANADA

SPACE PROGRAM

TITLE: Performance Analysis of the FTDCS Simulator & Operating System

AUTHOR(S): G. Ram
D. Bowen
B.A. Bowen

ISSUED BY CONTRACTOR AS REPORT NO: None

PREPARED BY: CompEng Serv Ltd.
Suite 600, 265 Carling Ave.
Ottawa, Ontario
K1S 2E1
(613) 563-1920

DEPARTMENT OF SUPPLY AND SERVICES CONTRACT NO: 36001-9-3593
S.N. 660ER-8-0003/39

DOC SCIENTIFIC AUTHORITY: J.M. Savoie

CLASSIFICATION: Unclassified

This report presents the views of the author(s). Publication of this report does not constitute DOC approval of the reports findings or conclusions. This report is available outside the department by special arrangement.

DATE: February 21, 1990

PREFACE

This work was performed for the Department of Communications under DSS Contract Number 36001-9-3593 entitled, "FTDCS Analysis".

This document describes the analysis of the FTDCS operating system and associated development environment.

The analysis methodology, conclusions and recommendations are provided in the main body of the report. The detailed results of the analysis are provided in the five Appendices.

COMMUNICATIONS CANADA
C R C
MAR 27 1990
LIBRARY – BIBLIOTHÈQUE

Table of Contents

Appendices

1. INTRODUCTION

This report describes the analysis of the Fault Tolerant Distributed Computer System (FTDCS) operating system and associated Sun-based simulator. The purpose was to reconstruct from the source code the logical structure, behavior, functionality and performance of the FTDCS software. In addition, as a secondary mission, an evaluation of the extensibility and modularity is required.

The work was performed by executing a reverse engineering algorithm, which involved examining the source code to reconstruct the functionality of software modules, thus creating a description of the system behavior using structure diagrams, tabulating the data flow between modules and finally determining performance from the code traversed and the data passed. As part of this detailed analysis, an evaluation of the design and its implementation was formulated.

The results of such an analysis produces a rather bulky data set. This data is recorded in Appendices relevant to each step in the analysis algorithm. The data, not only documents the analysis and provides quantifiable evidence for the conclusions, but are useful separately as a basis for extending the functionality of the system and as a basis for designing and predicting the performance of new applications programs.

The body of the report is presented in eight sections. Section 2 records the analysis algorithm, executed to produce these results.

Section 3 gives an overview of the FTDCS software as it is purported to have been designed.

Section 4, 5, & 6 contain the results of the execution of the input data set. Section 4 is devoted to a description of the software modules as identified from the examination of the source code. Section 5 is a behavioral description of the interaction of these modules based on the call hierarchy. Section 6 tabulates the data structures.

Section 7 contains an estimate of performance, Section 8 contains our evaluation of the overall design of the code. Section 9 contains a summary of the results with respect to the statement of work, and our conclusions and recommendations based on these results and our extensive exposure to the details of the implementation.

2. THE ANALYSIS ALGORITHM

The system was analyzed by executing the following algorithm:

1. Create the Analysis Data Base

   The data set supplied to the project team was examined to determine the purpose, functionality and implementation of the FTDCS software. The data set was organized to support the requirements of this algorithm.

2. Determine Software Modularity

   The source code was analyzed to determine the structure, extent and purpose of all the identifiable software modules, as well as the data stored and passed.

3. Determine Behavior

   System behavior was determined using structure diagrams based on the data obtained in step 2 which shows the calling hierarchy and the structure of the calls of all modules.

4. Determine Data Structures and Data flow

   From the data obtained in Step 2, a tabulation of all data structures was obtained.

5. Determine Performance

   From step 2 to 4, and in terms of the expectations in 1, an evaluation of the system design and the implementation is obtained; from steps 3 and 4, a performance analysis of the execution of all system calls is obtained.

3. AN OVERVIEW OF THE SYSTEM

The FTDCS is a fault tolerant distributed computer system designed to utilize clusters of processors in applications requiring reliable, high performance real-time computation. The three major components of the FTDCS architecture are:

1) hardware building blocks,
2) software building blocks and,
3) a layered operating system.

The FTDCS hardware building blocks are buses and processing units (PUs). The software building blocks are developed using high level languages.

The operating system is composed of three distinct layers - the kernel, the executive and the distributed system manager. The layered operating system not only provides the designer with hardware transparency, but also provides the necessary fault detection, isolation and recovery mechanisms. Further details about the operating system are given in Appendix A.

Application program development for the FTDCS is supported by a simulator for system development and testing. System development consists of 4 phases: system definition, system design, system implementation and system configuration. The FTDCS simulator provides support in all the areas of system development. The simulator is also useful for evaluating several fault tolerant techniques. Appendix B gives more details about system development, with an example.

4. SYSTEM SOFTWARE MODULES

From the input data set, the modules of the FTDCS operating system and the simulator were identified. For each module the following information was accumulated:

- name of the module,
- its parameters,
- a brief description on its purpose,
- the number of lines of "C" code in it,
- list of all the other modules it calls,
- its output, and
- a brief description on whether the module is completed and if the module algorithm complies with its code.

Appendix A contains a brief description and example of the system development phases. Generic modules are grouped together, with a tabulation of parameters.

Appendix B lists the software modules used in implementing the FTDCS operating system. The modules are grouped together based on the 3 different layers of the operating system - kernel, executive and distributed system manager.

5. BEHAVIORAL ANALYSIS

A behavioral analysis defines interaction between all the software modules. It usually relates the calls in a hierarchical decomposition and provides information not only on the topology of the interaction, but on the structure of the call.

Such an analysis uses the definition of modules obtained in Section 4, and is conveniently presented as a structure diagram.

Appendix C describes in detail, the purpose of having a structure design for a system, the structure diagram conventions used in this document and the behavior of FTDCS software based on our analysis.

6. DATA STRUCTURES

The data structures necessary to support the system, and the data passed during intermodule calls form an important part of a performance estimate. These data were identified and isolated during the examination of the source code (Section 4).

A complete data dictionary, which resulted from this analysis is contained in Appendix D.

7. SYSTEM PERFORMANCE ANALYSIS

Based on the FTDCS source code, and the resulting structure diagram (see Appendix C, & D), this section details the performance of the FTDCS. The performance of the various system software modules is tabulated in Appendix E. Each row of this table corresponds to a software module listed in Appendices A or B.

The following describes in detail how the data listed for each module was obtained.

Column 1: Name of the software module..

Column 2: Number of other modules called in order to execute this module.

Column 3: The sum of the total # of lines of "C" code in order to execute this module. This is the sum of the total number of lines of code in this module and the total number of lines of code in each of the called modules.

Column 4: The number of lines of "C" code required to execute the module in the worst case. A worst case scenario for a module represents the longest path through it. In order to calculate what percentage of the total code is executed in the worst case situation, the following was done.

Five modules were chosen at random: "st_sw_config", "st_config_exec", "add_resource", "K_cpu_executive" and "xnetwork_assign". It was assumed that the loops in these modules are executed only once (for any other number, the percentage needs to be modified accordingly). Next, the sum of the total number of lines of "C" code in each of the 5 modules was determined (205). Also the sum of the number of lines of code required to traverse the longest path through these 5 functions was calculated (170). The ratio of the latter to the former was about 83%. Hence, it was assumed that about 83% of the total code is run in the worst case. Thus the results in column 4 are obtained using:

0.83 * (result in column 3).

Column 5: The number of lines of assembly code required to execute the module in the worst case. This consists of both the actual code and the overhead in the conversion from "C" to assembly code.

The following assumptions are made for the calculations. A line of C code is approximately equal to 10 lines of assembly code. Also, for every C function called, about 30 lines of assembly code is added as overhead. Hence, the results in this column are obtained using:

[(result in column 4) * 10] + [(result in column 2) * 30].

Column 6: The number of clock cycles required to execute this module. For this, the following assumption is made. A line of assembly code is nearly equal to 6 clock cycles (for an Intel 8086). Hence, the results in this column are calculated using:

(result in column 5) * 6.

The tables in Appendix E give the time for executing most of the FTDCS functions in the worst case. These numbers can be used to choose the entry point functions in creating application programs with appropriate execution times.

8. DESIGN AND IMPLEMENTATION - AN EVALUATION

8.1 Comments on the System Design

The overall system can be commented on under two major heading: the design and the implementation.

The design appears to have been undertaken without a clear statement of requirements. These would have normally included, at least, a statement of the intended applications, the functionality and the performance. The logical design of the system would have addressed these requirements, in terms of the necessary functionality, and the overall organization of the necessary interfaces to the application programs. For example, exposing all layers of the system to the application programmer provides flexibility but leaves fault tolerance as a suspicious feature.

The implementation is incomplete with several missing modules, and the recursive nature of the Executive frustrates attaining the desired features.

8.2 Comments on Design Documentation

After constructing the tables for the FTDCS software modules, a detailed analysis of the FTDCS simulator and the operating system was done. This included constructing a tree of processes using the functions defined in the tables.

The following points are worth mentioning, with regard to the FTDCS software design, and related documentation.

1. A hierarchical approach is not maintained in the implementation of some of the modules. The code for these functions is scattered. This might probably have been due to the fact that a structure design was not done before implementation. Note however, that this does not mean that the module is incorrect.

   For example, one could implement the main simulator module as a loop which only calls other functions to execute chosen commands. Therefore, the clean up at the end of simulation would be left to the exit function. This would present a neater approach to understanding the working of the system. Currently, "s_test" does both.

2. It was mentioned by concerned authorities that some of the FTDCS software modules were not tested. However, the structure design assumes that not only all of them have been tested but also that they are all correct.

3. Some of the functions (e.g., KHI_bus, KHI_device and KHI_process) are implemented so that they rely on interrupts created by VMS.

4. Majority of the control entry point functions have not been implemented. However, there are provisions to add to them. The following is a list of functions which can be completed:

   K_link_control,
   xsimple_control and xnmr_control,
   kbus_control, kdevice_control and kprocess_control,
   xnetwork_control,
   kmb_master_control, kmb_slave_control and k188_control.

5. Since the documents claim that the system is based on an object oriented approach, it would be helpful if these objects (or data structures) are described as clearly and completely as possible. A bunch of files containing "C" code for the data structures is all that is available. Therefore, it is up to the reader to figure out why, how, and where these data structures are used.

   Also, there is no documentation on the purpose of the various fields in a data structure. For example, it is difficult to figure out the purpose of the field "sp_value" or "ds_value" in the data structure "context".

8.3 Comments on Source Code Documentation

The FTDCS source code listings and detailed design manuals for the simulator and the operating system were used to create tables of software modules.

During this study, the following shortcomings were noticed in the FTDCS documents (source code listings).

1. There is high level description of the system followed by a low level source code. How the transition from one to the other has been done is not explained anywhere. This makes it difficult to tie the two together and to check if the source code really satisfies system goals. It takes quite a while for an unfamiliar reader to understand the mapping.

2. A few of the data structures are only mentioned by name. Their contents are not defined.

3. Some functions have not been implemented (e.g., "mc_unknown_name"). Also, there is no documentation provided for the DSM fault manager modules and data structures.

4. The code and the algorithm for certain software modules have the following problems:

   * most often for a module, the algorithm explicitly states that a value is to be returned from it. But there is no value returned from the function in the source code.

   * a module source code may sometimes contain calls to other functions, and these calls are not described in the algorithm. Why these functions are called is thus not known. (Code lacks comments - see below).

   * there are no comments in the code whatsoever. This may not matter for simple functions. If functions are long and complicated (e.g., with nested loops and 'if' statements), it is difficult to determine what is happening or whether the code really does

what it is supposed to do. This is a very serious problem for someone wishing to modify the existing code. Note: it is very difficult for a non "C" programmer to read and understand functions with hundreds of lines of "C" code.

Also, algorithm statements in some functions are quite low level, simple and redundant (e.g., set X to 0). These statements are superfluous and more appropriate to appear as comments in the code.

* some complicated functions do not have an associated algorithm. What the module does or is supposed to do can only be determined from its brief (often 1 line) description. Therefore it is not possible to establish whether the module is correct.

* occasionally there are discrepancies between the algorithm and the code. For example, the algorithm may state "perform function X"; the code might contain the statement "if conditions A and B are true - then perform the function X". It is conditions like these that define the structure design of the module.

* order of function calls differ in certain modules. This may affect the behavior of the module and its end result. For example, the steps: square X and then increment X yield a result different from the result obtained by incrementing X and then squaring it.

* for some of the functions, the code does not contain the implementations for major portions of the algorithm (e.g., "xnmr_done").

8.4 Comments on Fault Tolerance

Since it is possible for application code to access the Operating System through the Distributed System Manager, the Executive or the Kernel, application programs can by-pass the Distributed System Manager, and thus the fault tolerance protection claimed by the overall system.

The Fault Tolerant features of the Operating System were not complete, as shown in Section 8.2.

The recursive nature of the Executive lends itself to programmer's abuse and errors.

The Distributed System Manager appears to have a single point failure. It would seem that for all processes there will be one master process on one CPU which spawns processes for execution on multiple machines, analyzes the results and determines sanity.

8.5 Comments on Performance

The Tables documented in the appendices provide timing data that can be used in determining the performance of a proposed application. The Executive portion is difficult to get timing for (because of its recursive nature).

For most of the operating system executive modules in the tables, no data is given. This is due to the fact that the function calls they make are very convoluted. It is quite impossible to determine the total number of lines of code in them, as it is very difficult to trace the more than hundred function calls made. Moreover, the number of lines of code in the worst case is very much execution dependent. Figure F1 shows an example of the function calls made by one such module "XB_boot".

Some of the Shell functions could take up to 250 milli seconds, which could seriously affect real time performance.

9 SUMMARY AND CONCLUSIONS

9.1 Summary

Under the terms of the contract, we have examined the FTDCS design documents and source code. In order to develop an understanding of the system, to model its performance, we have performed an extensive reverse engineering exercise. The results have been tabulated and presented in the appendices. We have also developed comprehensive tables which will allow performance analysis for any application program to be written in the future.

As well, the results in the first four appendices provide the designers data base for anyone tasked with adding to the existing software. It will be necessary for anyone working in the program to study these appendices and use them as a road map when changes must be made and to determine what the ripple effect of changes will be.

It is generally concluded that the Operating System is a prototype aimed at the distributed processing applications. It is hard to conceive of how and why one would use it as it is currently designed and built. Adding this prototype functions to the operating system functions of a real time operating system could create a more usable product.

The other major concern raised is the extensive use of recursion in programing the Executive. This use of recursion appears as a misuse of the capabilities provided by recursive programming techniques.

9.2 Conclusions

9.2.1 System Design

It is not apparent from either the supplied documentation or the analysis of the source code that a complete system requirements analysis and design was performed before beginning this project.

The high level description followed by pseudo code is inappropriate for Object Oriented Program design.

There is no explanation of how this operating system would be used by application programmers or if it provides sufficient functionality. A comparison to other real time operating systems such as VRTX or Harmony would identify many features for an application program, which are not available in this system.

As a result of the convoluted, recursive and undocumented design. It will be a very difficult job to modify this code. The number of other functions called and how modifications might affect them is hard to determine. This will make the maintenance costs high. Thus, it will be difficult for a third party to assume responsibility for life cycle support.

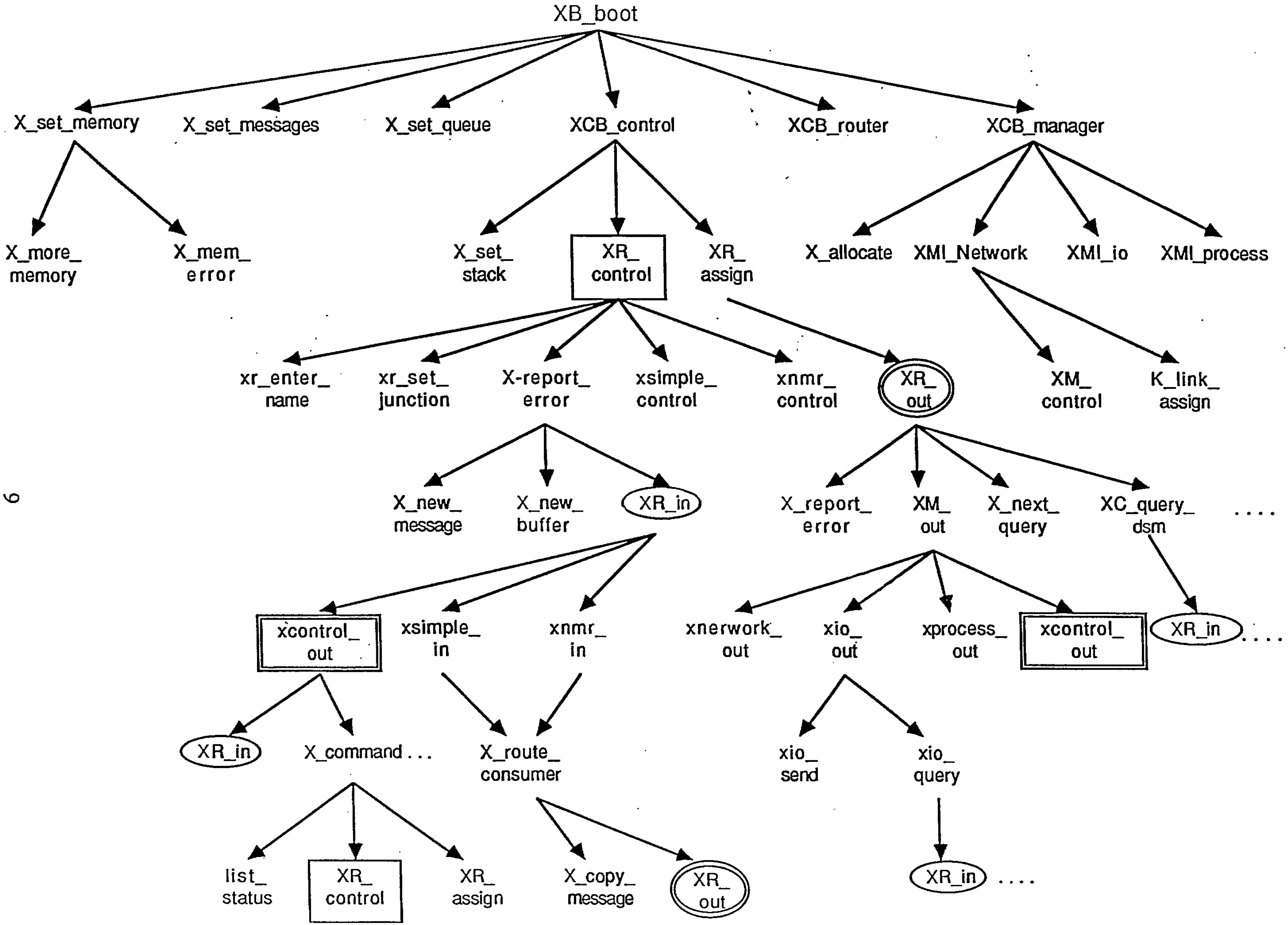


Figure F1: Possible execution paths for XB_boot

The design appears to have been more concerned with the Fault Tolerant issues than achieving real time performance and functionality.

9.2.2 Fault Tolerance

The operating system has been design around the intercommunication between consumers and users and between different layers of the operating system. The facility to have tasks executing on separate machines exist, however, the fault tolerant algorithm is incomplete. It does not have the control function for responding to detected faults implemented.

10. RECOMMENDATIONS

Future Enhancements

The following is a list of recommended actions assuming future work will be undertaken using this software.

1. Study of User needs and requirements: The operational environment of the FTDCS should be analyzed to determine any missing functionality. As well any special needs for distributed processing and fault tolerance should be defined.

2. Study of Features: The performance requirements, the fault tolerance requirements and the distributed nature of the operational environment.

3. A full identification of what modules have been tested and work: There is a question remaining of which modules work and which have not been tested. This should be resolved.

4. Implement missing functions: The functions listed above should be implemented.

5. Re Implement the Executive: Ideally the executive should be reimplemented to minimize the recursive nature and thread like style.

6. Add new input/output servers to Kernel (RS323, RS422, IEEE488, etc.): Any new I/O servers should be identified and added.

7. Modify as a result of User needs study: Depending on the users need study new functionality should be added to the operating system.

11. REFERENCES

1. FTDCS Applications: Source Code Listings, June 15, 1987, PRIOR Data Sciences Ltd., Kanata, Ontario, Canada.

2. FTDCS Applications: System Interface Design and Programmer's Guide, June 15, 1987, PRIOR Data Sciences Ltd., Kanata, Ontario, Canada.

3. FTDCS Operating System Revisions: Detailed Design, April 22, 1988, PRIOR Data Sciences Ltd., Kanata, Ontario, Canada.

4. FTDCS Operating System Revisions: Source Code Listings, April 22, 1988, PRIOR Data Sciences Ltd., Kanata, Ontario, Canada.

5. FTDCS Software Development: Source Code Listings, April 22, 1988, PRIOR Data Sciences Ltd., Kanata, Ontario, Canada.

6. FTDCS Software Development: System Programmer's Guide, April 22, 1988, PRIOR Data Sciences Ltd., Kanata, Ontario, Canada.

7. A Fault-Tolerant Distributed Computer System For Spacecraft and Other Applications, Executive Summary: May 31, 1987, Tridex Systems Inc., Nepean, Ontario, Canada.

APPENDIX A

TABLES OF SOFTWARE MODULES IN THE FTDCS SIMULATOR

(from 1988 manuals)

Table of Contents

List of Tables

1. INTRODUCTION

Developing a FTDCS system consists of four phases: definition, design, implementation and configuration. The FTDCS simulator provides support in all the four phases.

The following four sections describe the four system development phases supported by the simulator.

2. SYSTEMS DEFINITION

The entire system definitions consists of - hardware and software definitions. It specifies the hardware and software components needed to meet the system requirements.

The simulator provides an interactive interface through which the system hardware definition (specifications of processors, I/O devices, network devices and their interconnections) can be specified.

The software definitions can be specified through a text file, which can be edited through any interactive text editor. Data in the distributed software specification text file includes:

1. DSM consumer definition: text file entry is of the form:

   manager resource router processor [processor]

   where "resource" is the application resource which contains the "manager" (DSM) code, "router is the fault tolerant routing (FTR) algorithm used by DSM, and "processor" is the list of processors on which the DSM will run.

2. Application consumers definition: 3 types of entries for this:

   * Definition - consumer definition:

   define consumer resource router namecount

   where "define" is define consumer command, "consumer" is the name of the consumer to define, "resource" is the application or I/O resource for the consumer, "router" is FTR algorithm, and "namecount" is redundancy level for the consumer.

   * Linking - consumer output channels:

   link consumer branch0 [/ branch1 ... / branchN]

   where "link" is a link consumer command, "consumer" is consumer to link, "branchX" is a consumer name to which the output message will be sent.

   * Running - activating consumers at system startup time:

   run consumer

where "run" is the run consumer command and "consumer" is the name of the consumer to run at system startup time.

The system definition is used by the simulator to create a system model.

Functions:

The functions in Table 2.1 implement the system definition in order to be submitted to the DSM. The following functions are in the file stsystem.c.

Module	Parameters	Module Description
st_system	model	creates a simulator system definition
st_set_system	model	initializes system definition structure
st_sw_config	model	reads & interprets distributed s/w specs
read_consumers	file_buffer, cmd_buffer	reads application consumer specs & creates DSM commands
std_define	file_buffer, cmd_buffer	creates define consumer command
std_link	file_buffer, cmd_buffer	creates link consumer command
std_run	file_buffer, cmd_buffer	creates run consumer command
st_sys_config	model	interprets and integrates hardware model to the system definition
add_sys_manager	model, system, manager_id	adds a resource manager to system definition
add_sys_router	model, system, router_id	adds a routing manager to system definition
add_sys_resource	model, system, resource_id	adds a resource to system definition
add_sys_exec	model, system, cpu_id	adds a processor's local executive id to system definition

Table 2.1 Simulator Definition Functions.

Module	Lines in code	Calls made to	Return Value
st_system	7	st_set_system st_sw_config st_sys_config	none
st_set_system	47	none	none
st_sw_config	57	read_consumers	none
read_consumers	15	std_define std_link std_run	none
std_define	23	sh_define_cmd	none
std_link	35	sh_set_link_cmd sh_add_link_cmd	none
std_run	9	sh_run_cmd	none
st_sys_config	35	add_sys_manager add_sys_router add_sys_resource add_sys_exec	none
add_sys_manager	11	none	none
add_sys_router	12	none	none
add_sys_resource	15	none	none
add_sys_exec	22	none	none

Table 2.1 Simulator Definition Functions (Contd.)

Module	Completed?	Is code the exact implementation of algorithm?
st_system	yes	yes
st_set_system	yes	yes
st_sw_config	yes	algorithm does not quite explain the code
read_consumers	yes	yes
std_define	yes	algorithm does not quite explain the code
std_link	yes	algorithm does not quite explain the code
std_run	yes	algorithm does not quite explain the code
st_sys_config	yes	yes
add_sys_manager	yes	yes
add_sys_router	yes	yes
add_sys_resource	yes	yes
add_sys_exec	yes	yes

Table 2.1 Simulator Definition Functions (Contd.)

3. SYSTEMS DESIGN

The simulator provides two support aids for system design. These are in the form of information, displayed interactively during a simulation session. The two system design requirements are given below.

1. Loading Requirements: indicate which application software modules have to be compiled and linked for each processor. Command "show applications".
2. Local operating system requirements: indicate the kernel link servers and interrupt handlers to be incorporated into the local operating system kernel for each processor, i.e., shows which resources are linked to each processor. Command "show cpu cpuname".

Besides the simulator design, there are the distributed software (DSM and application software) design requirements and local operating system (kernel, configuration and executive) design requirements.

4. SYSTEM IMPLEMENTATION

System implementation consists of the implementation of the following modules:

1. Distributed Software implementation: consists of coding, compiling and linking software modules determined by the system design for - DSM and application software.
2. Local operating system executive implementation: consists of the implementation of the executive resource manager components and routing manager components. For each of these components, the implementation consists of coding the component initialization function and its associated event handling functions (e.g., "in", "out", "assign" and "control").
3. Local operating system kernel implementation: consists of the implementation of the local operating system kernel components - processor specific functions, kernel interrupt handlers and kernel link servers.

   The processor specific functions include the processor reset trap function and a number of kernel processor management functions. This code is written in assembly.

   Interrupt handlers service processor interrupts. Implementation of an interrupt handler requires the coding of at least 2 functions: an initialization function and an interrupt service routine.

   Kernel link servers perform the kernel level processing associated with individual processor resource links. Implementation of kernel link managers consists of coding the component initialization function and its associated event handling functions (e.g., "in", "out", "assign" and "control").
4. Local Configuration Specification: consists of data in a form which can be interpreted by the simulator to produce configuration (text) files for each local operating system. Local configuration specifications include data which defines interrupt handlers and kernel link servers for each processor. For example, shared memory addresses, code

segment and stack addresses and length, etc. Tables below show the modules for processor independent configuration and modules to interpret local configuration specs. for a simulated processor. These functions are defined in the file stconfig.c and stsimconfig.c

Module	Parameters	Module Description
st_config	model, cpu_id, config_fn	creates a local OS config structure based on system definition & local config. specs
st_config_header	model, cpu, config	sets up a header for a local executive configuration data structure
st_config_exec	model, cpu, config	creates DSM and executive portions of local configuration data structure
config_resources	model, cpu, config, exec_mark	adds executive resource data to the local configuration data structure
config_execs	model, home_cpu, config, exec_mark	adds linked executive data to the local configuration data structure
add_exec_consumer	model, exec_id, config, exec_mark link_id	adds executive consumer definition to the local configuration data structure
config_dsm	model, system, cpu, link_mark, config	adds DSM consumer to the local configuration data structure
st_simconfig	model, cpu, config	creates config data for processor's local OS running on the simulator
simconfig_managers	model, cpu, config	initializes exec. resource managers configuration data
simconfig_handlers	model, cpu, config	initializes kernel interrupt handlers configuration data
simconfig_servers	model, cpu, config	initializes kernel link servers configuration data
simconfig_resources	model, cpu, config	initializes processor's linked resources configuration data
simconfig_links	model, cpu, config	initializes processor's resource links configuration data
simconfig_kernel	model, cpu, config	adds kernel configuration data to the configuration data structure
simconfig_kmemory	model, cpu, config	adds kernel memory manager config. to the configuration data structure
simconfig_kcpu	model, cpu, config	adds kernel processor manager config. to the configuration data structure
simconfig_klink	model, cpu, config	adds kernel link manager config. to the configuration data structure
sim_network_link	model, link_id, config, marker	adds link to n/w resource config. to the configuration data structure
sim_io_link	model, link_id, config, marker	adds link to I/O resource config. to the configuration data structure
sim_process_link	model, link_id, config, marker	adds link to application process resource config. to the configuration data structure

Table 4.1 Local Configuration Specification Modules.

Module	Lines in code	Calls made to	Return Value
st_config	13	st_config_header config. entry point	none
st_config_header	11	none	none
st_config_exec	39	config_resources config_execs config_dsm	none
config_resources	50	none	none
config_execs	20	add_exec_consumer	none
add_exec_consumer	24	none	none
config_dsm	65	none	none
st_simconfig	13	simconfig_managers simconfig_handlers simconfig_servers simconfig_resources simconfig_links simconfig_kernel st_config_exec	none
simconfig_managers	14	none	none
simconfig_handlers	17	none	none
simconfig_servers	16	none	none
simconfig_resources	8	none	none
simconfig_links	22	none	none
simconfig_kernel	9	simconfig_kmemory simconfig_kcpu simconfig_klink	none
simconfig_kmemory	13	none	none
simconfig_kcpu	11	none	none
simconfig_klink	65	sim_network_link sim_io_link sim_process_link	none
sim_network_link	23	none	none
sim_io_link	18	none	none
sim_process_link	19	none	none

Table 4.1 Local Configuration Specification Modules (Contd.)

Module	Completed?	Is code the exact implementation of algorithm?
st_config	yes	yes
st_config_header	yes	yes
st_config_exec	yes	yes
config_resources	yes	yes
config_execs	yes	yes
add_exec_consumer	yes	yes
config_dsm	yes	code easier to follow with comments
st_simconfig	yes	yes
simconfig_managers	yes	yes
simconfig_handlers	yes	yes
simconfig_servers	yes	yes
simconfig_resources	yes	yes
simconfig_links	yes	yes
simconfig_kernel	yes	yes
simconfig_kmemory	yes	yes
simconfig_kcpu	yes	yes
simconfig_klink	yes	yes
sim_network_link	yes	yes
sim_io_link	yes	yes
sim_process_link	yes	yes

Table 4.1 Local Configuration Specification Modules (Contd.)

5. SYSTEM CONFIGURATION

The system configuration consists of 2 "C" files with data required to configure each local operating system. The 2 files are the functional configuration table file and the configuration data file.

1. Functional Configuration table: this is a table of entry points through which the local operating system functionality can be accessed at both the kernel and the executive levels. Entry points can be divided into 2 groups: configuration dependent entry points (initialization function) generated by the simulator according to local configuration specifications (Section 4.1); and entry points common to all local operating systems (e.g., boot functions).

2. Configuration Data: this is the data required to initialize the local operating system at processor boot time. The data consists of:

   (a) Configuration data header:

   - system processor id,
   - length of kernel, executive and DSM data components

   (b) Kernel configuration data:

- initialization data for kernel memory, processor and link managers
- initialization data for kernel interrupts and link servers
- hardware dependent data associated with each of the processor's resource links

(c) Executive configuration data:

- initialization data for executive controller and executive routing and resource managers

(d) Distributed system manager data: This data is included only if the DSM is scheduled on the processor for which the configuration data is being generated.

- string of commands describing the system software and hardware configuration to the DSM, generated from simulator's system definition.

Functions:

The simulator produces the local configuration data and the local functional configuration table using the system definition and the local configuration specification. The modules which provide this functionality are described in Table 5.1. They are defined in the file stfile.c.

Module	Parameters	Module Description
st_file	model, prefix, config_fn	produces config data & functional config. table
stf_data	name, config	produces configuration data file.
add_header	fid, name, header	writes configuration data header to configuration data file
add_kernel	fid, ptr	writes kernel configuration data to configuration data file
add_exec	fid, exec	writes executive configuration data to configuration data file
add_dsm	fid, ptr	writes DSM configuration data to configuration data file
add_dsm_command	fid, command	writes a DSM command to configuration data file
stf_table	name, config	produces a functional configuration table file
tbl_header	fid, name, header	writes configuration table header to the functional configuration table file
tbl_handlers	fid, config	writes handler init. entry points to the functional configuration table file
tbl_servers	fid, config	writes server init. entry points to the functional configuration table file

Table 5.1 Simulator Configuration Support Functions.

Module	Lines in code	Calls made to	Return Value
st_file	18	st_config stf_data stf_table	none
stf_data	26	add_kernel add_header add_exec add_dsm	none
add_header	31	none	none
add_kernel	13	none	none
add_exec	94	none	none
add_dsm	7	add_dsm_command	none
add_dsm_command	18	none	none
stf_table	18	tbl_header tbl_handlers tbl_servers	none
tbl_header	28	none	none
tbl_handlers	28	none	none
tbl_servers	28	none	none

Table 5.1 Simulator Configuration Support Functions (Contd.).

Module	Completed?	Is code the exact implementation of algorithm?
st_file	yes	yes
stf_data	yes	algorithm does not fully explain the code
add_header	yes	yes
add_kernel	yes	yes
add_exec	yes	code easier to read with comments
add_dsm	yes	yes
add_dsm_command	yes	yes
stf_table	yes	algorithm does not fully explain the code
tbl_header	yes	yes
tbl_handlers	yes	yes
tbl_servers	yes	yes

Table 5.1 Simulator Configuration Support Functions (Contd.).

For each process, a local operating system image is produced by compiling the functional configuration tables and linking it to the processor reset trap function.

After the generation of all system software, it is downloaded to the target according to the load specification known to the local operating system.

6. SYSTEM DEVELOPMENT EXAMPLE

6.1 Description

This section describes an example of system development. It lists the modules used in each phase of the system development (Sections 2-5).

The test system application provides a system which monitors two I/O devices, performs high priority processing of one and low priority processing of the other. In addition, there is an interface to DSM in order to issue commands to the operating system.

The test system hardware configuration consists of two processing sites connected via a multibus. One of the processors is also connected to an advanced communicating computer (also via a multibus), which is used for I/O device management.

6.2 Definition

Hardware Specification: six hardware components:

- 86/35 SBC (named k35)
- NIU processor (named kniu)
- console terminal (named console)
- I/O devices (named io1 and io2)
- multibus inter-processor link (named multibus)

Distributed Software Specification: describes to the simulator the software components of the system. The application processes include "dsm", "test1", "test2" and "shell". The software specification file for input to the simulator is given in detail in Section 7.2.2 in FTDCS Software Development - System Programmer's Guide. Commands: define, link and run are used to describe the specifications for the application processes.

6.3 Design

The system design requirements identify the software components required to implement the system at both the distributed software and local operating system levels.

The following simulator commands can be used: "show applications", "show cpu k35" and "show cpu kniu".

6.4 Implementation

6.4.1 Distributed Software Implementation

This consists of coding, compiling, linking and locating distributed software modules. This includes coding for "dsm" (described in 'Tables of Software Modules in FTDCS Operating System' & 'FTDCS Operating System Revisions'), "test1", "test2" and "shell".

The "shell" functions are the first group of functions in the table below. They are defined in the files: shell.c, shcommand.c and shstatus.c. These functions process operating system commands.

Application processes like "test1" and "test2" can send messages to and receive messages from the OS using the basic run time library functions. These functions are defined in the file rtllib.c and listed as the second set in the table below.

The application processes "test1" and "test2" are defined in the files test1.c and test2.c respectively. They are listed as the third set of functions in Table 6.1.

Module	Parameters	Module Description
main	none	mainline function for operating system shell
sh_define	command_buffer	processes "define consumer" shell commands
sh_link	command_buffer	processes "link consumer" shell commands
sh_run	command_buffer	processes "run consumer" shell commands
sys_command	command, ack_buffer ack_length	sends a command to the DSM
sh_status	buffer	processes "status" shell commands
consumer_status	buffer	processes "status consumer" shell commands
exec_status	buffer	processes "status cpu" shell commands
sys_accept	notify, reply, reply_length	accepts a query or call message (no block)
sys_query	branch, notify, parameter, data, data_length, reply, reply_length	sends a query message (no block)
sys_reply	consumer, signature, data, data_length	replies to a query or call message (no block)
sys_call	branch, data, data_length, reply, reply_length	sends a query message, waits for reply (block)
sys_receive	reply, reply_length	waits for data message (block)
sys_send	branch, data, data_length	sends a data message (no block)
sys_ready	none	sets ready state (block)
wait_event	mode	waits for an event
setup	none	sets up processor for simulator compatibility
tx_packet	none	sends packet to operating system
rx_packet	none	receives packet from operating system
test1	none	application process to process input data and send results to output channel 0
accept_fn	source, sign, data, length	replies to a query message with most recent input data
test2	none	application process to receive input data data from channel 0, and send it to channel 1
process_data	ptr1, l1, ptr2, l2, ptr3	combines two character strings to a third

Table 6.1 Distributed Software Implementation Functions.

Module	Lines in code	Calls made to	Return Value
main	27	sh_define sh_link sh_run sh_status U_set_memory U_set_command U_str_line U_flush_line	none
sh_define	17	sh_define_cmd sys_command U_str_line U_word_line	none
sh_link	36	sh_set_link_cmd sh_add_link_cmd sys_command U_str_line	none
sh_run	9	sh_run_cmd sys_command U_str_line	none
sys_command	16	sys_write sys_call	length of data returned in acknowledgement buffer
sh_status	20	exec_status consumer_status U_flush_line U_str_line	none
consumer_status	48	sys_call U_flush_line U_str_line	none
exec_status	63	sys_call U_flush_line U_str_line	none

Table 6.1 Distributed Software Implementation Functions.

Module	Lines in code	Calls made to	Return Value
sys_accept	12	tx_packet	none
sys_query	19	tx_packet	none
sys_reply	13	tx_packet	none
sys_call	17	tx_packet wait_event	call reply data length
sys_receive	12	tx_packet wait_event	received data length
sys_send	11	tx_packet	none
sys_ready	7	tx_packet wait_event	none
wait_event	36	tx_packet rx_packet	none
setup	7	none	none
tx_packet	6	none	none
rx_packet	6	none	none
test1	19	setup sys_receive sys_send sys_accept sys_reply accept_fn	none
accept_fn	10	sys_reply sys_accept	UPKT_READY, indicating return to ready state
test2	24	setup sys_receive sys_call sys_send process_data	none
process_data	18	none	concatenated string length

Table 6.1 Distributed Software Implementation Functions (Contd.).

Module	Completed?	Is code the exact implementation of algorithm?
main	yes	yes (function has no "quit" - infinite loop)
sh_define	yes	yes
sh_link	yes	algorithm does not fully explain the code
sh_run	yes	yes
sys_command	yes	yes
sh_status	yes	yes
consumer_status	yes	code easier to follow with comments
exec_status	yes	code easier to follow with comments
sys_accept	yes	yes
sys_query	yes	yes
sys_reply	yes	yes
sys_call	yes	yes
sys_receive	yes	yes
sys_send	yes	yes
sys_ready	yes	yes
wait_event	yes	yes
setup	yes	yes
tx_packet	yes	yes
rx_packet	yes	yes
test1	yes	yes
accept_fn	yes	yes
test2	yes	yes
process_data	yes	yes

Table 6.1 Distributed Software Implementation Functions (Contd.).

6.4.2 Kernel Implementation

The kernel implementation consists of processor specific and kernel interrupt handlers and link server functions.

The processor specific functions are described below:

- k86cpu.c contains kernel processor manager functions. They are similar to the functions listed in Table 2.1 of Appendix B. The difference being these functions are more hardware specific than the functions in the tables. They make calls to assembly language routines required specifically for the operating system kernel on an Intel 8086. For example, K_cpu_kernel calls K86_DISABLE instead of the general KU_disable. The algorithms and code however remain the same. The assembly language routines are defined in the file "k86mc.asm".

- k86memory.c contains the kernel memory manager functions. They are the same as the functions in Table 2.2 of Appendix B.

- k86link.c contains the kernel link manager functions. They are the same as the functions in Table 2.3 of Appendix B.

Following are the kernel interrupt handlers and link servers functions:

- the file k86process.c contains applications to system server functions. They are similar to the functions in "ksimprocess.c" listed in Table 2.4 of Appendix B. However, the following differences can be listed between the two sets of functions: functions in "k86process.c" are more hardware specific (and use the assembly language routines given in the file "k86prmc.asm"), function KHI_process (the interrupt handler function) is implemented.

- functions in kmbmaster.c are multibus shared memory master server functions. The C functions are listed as the first group of functions in Table 6.2 below. The related assembly language routines are defined in the file "kmbmasmc.asm".

- functions in kmbslave.c are multibus shared memory slave server functions. The C functions are listed as the second group of functions in Table 6.2 below. The related assembly language routines are defined in the file "kmbslvmc.asm".

- k188.c contains functions which implement a server to support the 188/48 I/O processors. The C functions are listed as the third group of functions in Table 6.2 below. The related assembly language routines are defined in the file "k188mc.asm".

Module	Parameters	Module Description
KSI_mb_master	server	server initialization entry point
KHI_mb_master	unit_base, vector unit_count	interrupt handler initialization entry point
kmb_master_in	server, status, bus_id	server in entry point
kmb_master_out	server, bus_id, packet	server output entry point
kmb_master_assign	server, link, name	server assign entry point
kmb_master_control	server, code, local_id, data	server control entry point
KSI_mb_slave	server	server initialization entry point
KHI_mb_slave	unit_base, vector unit_count	interrupt handler initialization entry point
kmb_slave_in	server, status, bus_id	server in entry point
kmb_slave_out	server, bus_id, packet	server output entry point
kmb_slave_assign	server, link, name	server assign entry point
kmb_slave_control	server, code, local_id, data	server control entry point
KSI_i188	server	server initialization entry point
KHI_i188	unit_base, vector unit_count	interrupt handler initialization entry point
k188_in	server, data, tty_id	server in entry point
k188_out	server, tty_id, packet	server output entry point
k188_assign	server, link, name	server assign entry point
k188_control	server, code, local_id, data	server control entry point
k188_receive	tty, offset, length	processes receive interrupt
in_188_raw	tty, e_ptr	processes input characters
k188_transmit	tty	processes transmit complete interrupt
out_188_raw	tty, data, length	processes output characters
k188_tx_packet	tty, data, length	sends characters to 188/48 board

Table 6.2 Resource Link Specific Functions.

Module	Lines in code	Calls made to	Return Value
KSI_mb_master	19	K_allocate	none
KHI_mb_master	7	MB86_M_INIT	none
kmb_master_in	49	K_new_network K_new_buffer K_cpu_k2x K_free_network K_release_buffer K_copy_buffer K_next_queue MB86_SIG_SLAVE	none
kmb_master_out	22	K_add_queue K_new_network K_copy_buffer MB86_SIG_SLAVE	none
kmb_master_assign	30	K_allocate K_new_network K_new_buffer MB86_M_CONTROL K_link_control	none
kmb_master_control	11	device specific control functions	none
KSI_mb_slave	19	K_allocate	none
KHI_mb_slave	7	MB86_S_INIT	none
kmb_slave_in	40	K_new_network K_new_buffer K_cpu_k2x K_free_network K_release_buffer K_next_queue MB86_SIG_MASTER MB86_SLAVE_RX MB86_PUT_S2M MB86_SLAVE_TX MB86_GET_M2S	none
kmb_slave_out	19	K_add_queue K_new_network K_copy_buffer MB86_PUT_S2M MB86_SIG_MASTER	none
kmb_slave_assign	30	K_allocate K_new_network K_new_buffer MB86_S_CONTROL K_link_control	none
kmb_slave_control	11	device specific control functions	none

Table 6.2 Resource Link Specific Functions.

Module	Lines in code	Calls made to	Return Value
KSI_i188	20	K_allocate	none
KHI_i188	7	I188INIT	none
k188_in	19	k188_receive k188_transmit I188IN	none
k188_out	18	out_188_raw K_new_io K_copy_buffer K_add_queue	none
k188_assign	50	K_allocate K_new_io K_new_buffer K_link_control I188SETUP I188OUT	none
k188_control	11	device specific control functions	none
k188_receive	18	in_188_raw k188_tx_packet I188DIN I188OUT	none
in_188_raw	37	K_cpu_k2x K_new_io K_new_buffer	number of characters in echo buffer
k188_transmit	18	in_188_raw k188_tx_packet K_next_queue out_188_raw K_release_buffer K_free_io	none
out_188_raw	20	k188_tx_packet K_new_buffer K_release_buffer	none
k188_tx_packet	13	I188DOUT I188OUT	none

Table 6.2 Resource Link Specific Functions (Contd.).

Module	Completed?	Is code the exact implementation of algorithm?
KSI_mb_master	yes	yes
KHI_mb_master	yes	yes
kmb_master_in	yes	code easier to read with comments
kmb_master_out	yes	yes
kmb_master_assign	yes	yes
kmb_master_control	yes	no, currently, control functions not implemented
KSI_mb_slave	yes	yes
KHI_mb_slave	yes	yes
kmb_slave_in	yes	code easier to read with comments
kmb_slave_out	yes	yes
kmb_slave_assign	yes	yes
kmb_slave_control	yes	no, currently, control functions not implemented
KSI_i188	yes	yes
KHI_i188	yes	yes
k188_in	yes	yes
k188_out	yes	yes
k188_assign	yes	code easier to read with comments
k188_control	yes	no, currently, control functions not implemented
k188_receive	yes	yes
in_188_raw	yes	yes
k188_transmit	yes	yes
out_188_raw	yes	yes
k188_tx_packet	yes	yes

Table 6.2 Resource Link Specific Functions (Contd.).

6.4.3 Executive Implementation

All components required to implement the executive are given in "FTDCS Operating System Revisions" manuals and listed in Appendix B. There are no changes made to these modules. The code modules are compiled and added to the executive code library.

6.4.4 Local Configuration Specification

The local configuration specification specifies to the simulator the kernel configuration data for each processor. The interrupt handlers and link servers data definitions are included. The local configuration specification for the processors k35 and kniu are given in detail in "FTDCS Software Development - System Programmer's Guide".

6.5 Configuration

System configuration consists of: the interpretation by the simulator of the local configuration specifications, generation of the local OS using the local configuration specifications, and downloading of all the system software to the target hardware.

The configuration modules are defined in the file st86config.c. They create the configuration data structures for an Intel 8086 family processor's local operating system. The functional configuration tables and the configuration data files thus created for the example are given in "FTDCS Software Development: System Programmer's Guide".

Module	Parameters	Module Description
st86_config	model, cpu, config	creates config data for Intel 8086's local OS running on the simulator
st86_managers	config	initializes exec. resource managers configuration data
st86_handlers	config	initializes kernel interrupt handlers configuration data
st86_servers	config	initializes server config. data structures
st86_resources	config, model	initializes resource config. data structures
st86_read	file, model, config	reads local configuration specification file
add_handler	file_buffer, config	reads a handler definition from the local configuration specification file
add_server	file_buffer, config	reads a server definition from the local configuration specification file
add_resource	file_buffer, config, model	reads a resource definition from the local configuration specification file
add86_network	file_buffer, resource	reads network resource specific data from local configuration specification file
add86_io	file_buffer, resource	reads I/O resource specific data from local configuration specification file
add86_process	file_buffer, resource	reads appl. process resource specific data from local configuration specification file
st86_links	model, cpu, config	initializes processor's resource links configuration data
st86_kernel	model, cpu, config	adds kernel configuration data to the configuration data structure
st86_kmemory	model, cpu, config	adds kernel memory manager config. to the configuration data structure
st86_kcpu	model, cpu, config	adds kernel processor manager config. to the configuration data structure
st86_klink	model, cpu, config	adds kernel link manager config. to the configuration data structure
st86_network_link	model, link_id, config, marker, resource	adds link to n/w resource config. to the configuration data structure
st86_io_link	model, link_id, config, marker resource	adds link to I/O resource config. to the configuration data structure
st86_process_link	model, link_id, config, marker, resource	adds link to application process resource config. to the configuration data structure

Table 6.4 Simulator Configuration Modules.

Module	Lines in code	Calls made to	Return Value
st86_config	23	st86_managers st86_handlers st86_servers st86_resources st86_read st86_links st86_kernel st_config_exec	none
st86_managers	12	none	none
st86_handlers	11	none	none
st86_servers	11	none	none
st86_resources	12	none	none
st86_read	17	add_handler add_server add_resource	none
add_handler	30	none	none
add_server	34	none	none
add_resource	61	add86_network add86_io add86_process	none
add86_network	17	none	none
add86_io	15	none	none
add86_process	29	none	none
st86_links	32	none	none
st86_kernel	9	st86_kmemory st86_kcpu st86_klink	none
st86_kmemory	13	none	none
st86_kcpu	11	none	none
st86_klink	65	st86_network_link st86_io_link st86_process_link	none
st86_network_link	24	none	none
st86_io_link	22	none	none
st86_process_link	26	none	none

Table 6.4 Simulator Configuration Modules (Contd.).

Module	Completed?	Is code the exact implementation of algorithm?
st86_config	yes	algorithm does not fully explain the code
st86_managers	yes	yes
st86_handlers	yes	yes
st86_servers	yes	yes
st86_resources	yes	yes
st86_read	yes	yes
add_handler	yes	algorithm does not fully explain the code
add_server	yes	algorithm does not fully explain the code
add_resource	yes	algorithm does not fully explain the code, also code is easier to read with comments
add86_network	yes	algorithm does not fully explain the code
add86_io	yes	algorithm does not fully explain the code
add86_process	yes	algorithm does not fully explain the code
st86_links	yes	yes
st86_kernel	yes	yes
st86_kmemory	yes	yes
st86_kcpu	yes	yes
st86_klink	yes	code is easier to read with comments
st86_network_link	yes	yes
st86_io_link	yes	yes
st86_process_link	yes	yes

Table 6.4 Simulator Configuration Modules (Contd.).

APPENDIX B

TABLES OF SOFTWARE MODULES IN THE FTDCS OPERATING SYSTEM

(from 1988 manuals)

Table of Contents

List of Tables

1. INTRODUCTION

The following tables lists the software modules used in implementing the FTDCS operating system. The modules are taken from the FTDCS Operating System - Detailed Design and Source Code Listings, April 1988. The operating system is composed of three distinct layers - the kernel, the executive and the distributed system manager.

The three sections below list the software modules used to implement them. The modules are grouped together according to the functions they perform. For example, all kernel memory management functions are listed in a table.

The information for each module is given in three tables. The first table gives the module name, its parameters and description. The second table lists the length of the code in number of lines, functions called by this module and return value for the module. The final table indicates if the module has been completed and whether the code is the exact implementation of the algorithm (and if not, what is the difference).

2. KERNEL DATA STRUCTURES & FUNCTIONS

The kernel is the machine dependent portion of the operating system specific to a single processor. It provides an interface between the operating system and the actual hardware. The three components of the kernel are processor manager, memory manager and link manager. The data structures and functions for processor management, memory management and interface link management are described in the following subsections.

2.1 Processor Management

Data Structures:

1. Flag indicating the mode of operation:

   - kernel mode,
   - executive mode,
   - user mode and
   - idle mode

2. Flags indicating the pending mode of operation of the operating system:

   - kernel mode,
   - executive mode and
   - user mode.

3. Event queues: 2 queues for each of the executive and kernel modes - a queue for events that are currently being processed and another for pending events.

4. A pointer to current application process context.

5. A table of kernel entry points available to the executive.

Functions:

The following functions are in the file ksimcpu.c.

Module	Parameters	Module Description
KB_boot	none	kernel boot entry
KMB_cpu	data	initializes kernel processor manager
K_cpu_kernel	none.	kernel mode operation
K_cpu_executive	kcpu	executive mode operation
K_cpu_enter	none	context switch on interrupt
K_cpu_exit	none	continue interrupted execution
K_cpu_fork	unit_id, data	pending switch from interrupt to kernel mode
K_cpu_k2x	xid, data	pending switch from kernel to executive mode
K_cpu_x2k	entry_id, data local	switch from executive to kernel mode
K_cpu_x2u	user_context	pending switch to user mode
K_cpu_u2x	none	block switch to user mode
KU_enable	none.	enables processor inputs
KU_disable	none.	disables processor interrupts
U_copy	from, to, length	performs utility copy
KU_invalid	none	identifies invalid function entry points

Table 2.1 Processor Management Functions

Module	Lines in code	Calls made to	Return Value
KB_boot	21	K_new_buffer K_release_buffer K_cpu_k2x K_cpu_exit	none
KMB_cpu	27	K_allocate	none
K_cpu_kernel	25	K_link_in K_cpu_executive KU_disable KU_enable	none
K_cpu_executive	27	KU_enable KU_disable	none
K_cpu_enter	10	none	none
K_cpu_exit	24	KU_disable KU_enable K_cpu_kernel	none
K_cpu_fork	14	none	none
K_cpu_k2x	13	none	none
K_cpu_x2k	14	K_cpu_kernel KU_disable KU_enable	none
K_cpu_x2u	8	none	none
K_cpu_u2x	6	none	none
KU_enable	4	none.	none
KU_disable	4	none.	none
U_copy	7	none.	none
KU_invalid	4	none.	none

Table 2.1 Processor Management Functions (Contd.)

Module	Completed?	Is code exact implementation of algorithm?
KB_boot	yes	K_release_buffer listed in called functions, but not called in code.
KMB_cpu	yes	yes
K_cpu_kernel	yes	yes
K_cpu_executive	yes	yes
K_cpu_enter	yes	yes
K_cpu_exit	yes	yes
K_cpu_fork	yes	yes
K_cpu_k2x	yes	yes
K_cpu_x2k	yes	yes
K_cpu_x2u	yes	yes
K_cpu_u2x	yes	yes
KU_enable	yes	algorithm not defined, but OK.
KU_disable	yes	algorithm not defined, but OK.
U_copy	yes	algorithm not defined, but OK.
KU_invalid	yes	algorithm not defined, but OK.

Table 2.1 Processor Management Functions (Contd.)

2.2 Memory Management

Data Structures:

Pointers to:

- general memory management structure,
- utility buffer management structure,
- packet management structures and
- utility queue management structure.

Functions:

The following functions are defined in the file ksimmemory.c. The functions are divided into five groups in the tables: kernel initialization, memory management, queue management, buffer and packet management, and executive accessible functions.

Module	Parameters	Module Description
KMB_memory	data	kernel memory manager initializer entry point.
K_refill	ptr, size	expands kernel memory pool
K_mem_error	ptr, code	kernel memory management error function
K_allocate	size	allocates kernel memory
K_reallocate	from_ptr, size	reallocates kernel memory
K_free	ptr	frees kernel memory
K_set_stack	size, initial, expand_size	preallocates blocks of kernel memory
K_add_queue	queue, data	adds item to queue tail
K_next_queue	queue	removes item from queue head
K_new_buffer	size	allocates a data buffer
K_release_buffer	buffer	releases a data buffer
K_free_buffer	buffer	deallocates a data buffer
K_copy_buffer	buffer	copies a data buffer
K_new_network	none	allocates network data packet
K_free_network	network_pkt	deallocates network data packet
K_new_io	none	allocates I/O consumer packet
K_free_io	io_pkt	deallocates I/O consumer packet
K_new_user	none	allocates application consumer packet
K_free_user	user_pkt	deallocates application consumer packet
X2K_allocate	size, ptr	allocates kernel memory from executive
X2K_free	not_used, ptr	frees kernel memory from executive
X2K_new_buffer	size, ptr	allocates data buffer from executive
X2K_free_buffer	not_used, ptr	deallocates data buffer from executive
X2K_free_network	not_used, ptr	deallocates network packet from executive
X2K_free_io	not_used, ptr	deallocates I/O packet from executive
X2K_free_user	not_used, ptr	deallocates user packet from executive

Table 2.2 Memory Management Functions

Module	Lines in code	Calls made to	Return Value
KMB_memory	23	K_set_stack	none
K_refill	10	none	none
K_mem_error	6	none	none
K_allocate	7	none	none
K_reallocate	10	none	none
K_free	5	none	none
K_set_stack	10	none	ptr to preallocated block control structure
K_add_queue	6	none	none
K_next_queue	7	none	ptr to queue head item data
K_new_buffer	8	none	ptr to allocated buffer
K_release_buffer	7	K_free_buffer	none
K_free_buffer	5	none	none
K_copy_buffer	6	none	input parameter
K_new_network	6	none	ptr. to network data packet
K_free_network	5	none	none
K_new_io	6	none	ptr. to I/O data packet
K_free_io	5	none	none
K_new_user	6	none	ptr. to user data packet
K_free_user	5	none	none
X2K_allocate	6	K_allocate	none
X2K_free	6	K_free	none
X2K_new_buffer	6	K_new_buffer	ptr. to allocated buffer
X2K_free_buffer	6	K_free_buffer	none
X2K_free_network	6	K_free_network	none
X2K_free_io	6	K_free_io	none
X2K_free_user	6	K_free_user	none

Table 2.2 Memory Management Functions (Contd.)

Module	Completed?	Is code exact implementation of algorithm?
KMB_memory	yes	yes
K_refill	yes	no algorithm defined for the module
K_mem_error	yes	no algorithm defined for the module
K_allocate	yes	no algorithm defined for the module
K_reallocate	yes	no algorithm defined for the module
K_free	yes	no algorithm defined for the module
K_set_stack	yes	no algorithm defined for the module
K_add_queue	yes	no algorithm defined for the module
K_next_queue	yes	no algorithm defined for the module
K_new_buffer	yes	no algorithm defined for the module
K_release_buffer	yes	no algorithm defined for the module
K_free_buffer	yes	no algorithm defined for the module
K_copy_buffer	yes	no algorithm defined for the module
K_new_network	yes	no algorithm defined for the module
K_free_network	yes	no algorithm defined for the module
K_new_io	yes	no algorithm defined for the module
K_free_io	yes	no algorithm defined for the module
K_new_user	yes	no algorithm defined for the module
K_free_user	yes	no algorithm defined for the module
X2K_allocate	yes	no algorithm defined for the module
X2K_free	yes	no algorithm defined for the module
X2K_new_buffer	yes	no algorithm defined for the module
X2K_free_buffer	yes	no algorithm defined for the module
X2K_free_network	yes	no algorithm defined for the module
X2K_free_io	yes	no algorithm defined for the module
X2K_free_user	yes	no algorithm defined for the module

Table 2.2 Memory Management Functions (Contd.).

2.3 Link Management

Data Structures:

1. Interrupt management data: a table of interrupting sources vs. kernel identifiers.

2. Link server management data: table of control structures for each kernel link server. Each entry consists of:

   - server ids,
   - server link count,
   - server interrupt, output, assign and control entry pointe,
   - ptr. to local server data

3. Link configuration data - a table of link configuration data for each kernel link.

4. Kernel identifier tables - a table of kernel ids vs. server and local ids.

Functions:

The following functions are defined in the file ksimlink.c.

Module	Parameters	Module Description
KMB_link	data	initializes kernel link manager
K_link_in	unit, data	link manager interrupt entry point
K_link_out	kid, data	link manager output entry point
K_link_assign	link_id, name	link manager assign resource entry point
K_link_control	code, kid, data	link manager control function entry point
kl_enter	name	adds link table entry

Table 2.3 Link Management Functions.

Module	Lines in code	Calls made to	Return Value
KMB_link	56	K_allocate, link handle init. & link server init. entry pts.	none
K_link_in	12	l.s. in entry pts.	none
K_link_out	12	l.s. out entry pts	none
K_link_assign	14	l.s. assign entry pts	none
K_link_control	10	kl_enter, link server control entry pts.	none none
kl_enter	27	none	none

Table 2.3 Link Management Functions (Contd.).

Module	Completed?	Is code exact implementation of algorithm?
KMB_link	yes	code definitely needs some comments
K_link_in	yes	slight discrepancy between code and alg.
K_link_out	yes	value returned in code but not in alg.
K_link_assign	yes	value returned in code but not in alg.
K_link_control	yes	algorithm not fully implemented
kl_enter	yes	K_reallocate called but not listed.

Table 2.3 Link Management Functions (Contd.).

2.4 Kernel Link Server Functions

The following functions are grouped together in the order of the files in which they are found: ksimbus.c, ksimio.c and ksimprocess.c. Ksimbus.c, ksimio.c and ksimprocess.c contain simulated network server, I/O server and application to system server functions respectively.

Module	Parameters	Module Description
KSI_bus	server	server initialization entry point
KHI_bus	unit_base, vector unit_count	handler initialization entry point
kbus_in	server, data, bus_id	server interrupt service entry point
kbus_out	server, bus_id, packet	server output entry point
kbus_assign	server, link, name	server assign resource entry point
kbus_control	server, code, local_id, data	server control function entry point
KSI_device	server	server initialization entry point
KHI_device	unit_base, vector unit_count	handler initialization entry point
kdevice_in	server, data, device_id	server interrupt service entry point
kdevice_out	server, packet, device_id	server output entry point
kdevice_assign	server, link, name	server assign resource entry point
kdevice_control	server, code, local_id, data	server control function entry point
KSI_process	server	server initialization entry point
KHI_process	unit_base, vector unit_count	handler initialization entry point
kprocess_in	server, packet not_used	server request entry point
kprocess_out	server, packet, process_id	server reply entry point
kprocess_assign	server, link, name	server assign resource entry point
kprocess_control	server, code, local_id, data	server control function entry point

Table 2.4 Kernel Link Server Functions.

Module	Lines in code	Calls made to	Return Value
KSI_bus	19	K_allocate	none
KHI_bus	6	none	none
kbus_in	22	K_new_network K_new_buffer K_cpu_k2x	none
kbus_out	14	none	none
kbus_assign	26	K_allocate K_link_control	none
kbus_control	13	device specific control functions	none
KSI_device	19	K_allocate	none
KHI_device	6	none	none
kdevice_in	21	K_new_io K_new_buffer K_cpu_k2x	none
kdevice_out	10	none	none
kdevice_assign	26	K_allocate K_link_control	none
kdevice_control	13	device specific control functions	none
KSI_process	22	K_allocate	none
KHI_process	6	none	none
kprocess_in	23	K_new_user K_new_buffer K_cpu_k2x K_cpu_u2x	none
kprocess_out	26	K_cpu_x2u K_cpu_u2x	none
kprocess_assign	37	K_allocate K_link_control	none
kprocess_control	13	device specific control functions	none

Table 2.4 Kernel Link Server Functions (Contd.).

Module	Completed?	Is code exact implementation of algorithm?
KSI_bus	yes	yes
KHI_bus	yes	null function as simulated interrupts created by VMS
kbus_in	yes	yes
kbus_out	yes	yes
kbus_assign	yes	yes
kbus_control	yes	discrepancy between code and algorithm, also value returned in code but not in algorithm
KSI_device	yes	yes
KHI_device	yes	null function as simulated interrupts created by VMS
kdevice_in	yes	yes
kdevice_out	yes	yes
kdevice_assign	yes	yes
kdevice_control	yes	discrepancy between code and algorithm, also value returned in code but not in algorithm
KSI_process	yes	yes
KHI_process	yes	null function as simulated interrupts created by VMS
kprocess_in	yes	yes
kprocess_out	yes	yes
kprocess_assign	yes	yes
kprocess_control	yes	discrepancy between code and algorithm, also value returned in code but not in algorithm

Table 2.4 Kernel Link Server Functions (Contd.).

3. EXECUTIVE DATA STRUCTURES & FUNCTIONS

The operating system executive layer is composed of 3 components. They are the executive controller, the executive routing manager, and the executive resource manager. The following subsections list the data structures and functions used to implement them.

3.1 Executive Controller

Data Structures:

1. Executive identification.

2. DSM routing data: specification of DSM consumer with which executive communicates.

3. Query support data: data to be used in querying DSM (same structures as DSM replies).

4. Executive controller entry points: invoked when messages are sent to or received from DSM.

Functions:

There are 2 groups of functions defined below. The first group of functions are the executive control functions found in the file xcontrol.c. The other group of functions defined in xutility.c and they are the executive utility functions. These functions are further divided into 3 groups: executive memory management, executive queue management and executive message and packet management functions.

Module	Parameters	Module Description
XB_boot	data	executive boot entry point
XCB_control	data	control component boot entry point
xcontrol_in	command	entry point from kernel
xcontrol_out	message	entry pt. to receive message to executive
X_command	message, command	processes executive command
list_status	query	reports executive status
XC_next_query	code, length, dest., local_id	allocates a system query command
XC_query_dsm	query, notify, parameter	requests data from DSM
X_report_error	code, error msg.	reports fault to DSM
X_set_memory	size	initializes executive memory management
X_more_memory	ptr, size	expands executive memory pool
X_mem_error	ptr, code	executive memory management error fn.
X_allocate	size	allocates executive memory
X_reallocate	ptr, size	reallocates executive memory
X_free	ptr	frees executive memory
X_set_stack	size, initial, expand	preallocates block of executive memory
X_set_queue	initial, expand	initializes executive queue management
X_add_queue	queue, data	adds item to queue tail
X_next_queue	queue	returns item from queue head
X_join_queue	from_queue, to_queue, join_queue	joins a queue to the tail of another queue
X_find_signature	queue, signature	searches a queue for a message signature
X_set_messages	initial, expand	initializes executive message mgmt.
X_new_message	none	allocates an executive message
X_free_message	message	deallocates an executive message
X_copy_message	message, dest.	copies an executive message
X_new_buffer	size	allocates a data buffer
X_free_buffer	buffer	deallocates a data buffer
X_copy_buffer	buffer	copies a data buffer
X_free_network	network_pkt	deallocates a network data packet
X_free_io	io_pkt	deallocates a I/O consumer packet
X_free_user	user_pkt	deallocates an application consumer packet

Table 3.1 Executive Controller Functions.

Module	Lines in code	Calls made to	Return Value
XB_boot	16	X_set_memory X_set_messages X_set_queues exec. components init. entry pts.	none
XCB_control	40	X_set_stack exec. router control entry pts. exec. router assign entry pts	none
xcontrol_in	15	X_new_message exec. manager out entry pts.	none
xcontrol_out	32	X_find_signature X_report_error X_free_buffer X_free_message X_copy_buffer X_command query notify fn. entry pts., exec. router in entry pts	none
X_command	32	X_command list_status X_report_error exec. router assign entry pts exec. router control entry pts	none
list_status	31	X_new_buffer X_new_message exec. router in entry pts	none
XC_next_query	14	X_new_buffer	ptr. to allocated system command
XC_query_dsm	23	X_new_message X_add_queue exec. router in entry pts	none
X_report_error	29	X_new_message X_new_buffer exec. router in entry pts	none
X_set_memory	8	X_more_memory X_mem_error	ptr to memory management structure
X_more_memory	6	none	none
X_mem_error	6	none	none
X_allocate	9	none	ptr. to allocated memory
X_reallocate	8	none	ptr. to reallocated memory
X_free	5	none	none
X_set_stack	9	none	ptr. to preallocated block control structure

Table 3.1 Executive Controller Functions (Contd.).

Module	Lines in code	Calls made to	Return Value
X_set_queue	8	none	ptr. to Q mgmt. structure
X_add_queue	6	none	none
X_next_queue	7	none	ptr. item data
X_join_queue	7	none	none
X_find_signature	26	none	none
X_set_messages	8	none	ptr. to message mgmt. control structure
X_new_message	9	none	ptr. to executive message
X_free_message	5	none	none
X_copy_message	11	X_copy_buffer	ptr. to copied exec message
X_new_buffer	7	none	ptr. to allocated buffer
X_free_buffer	9	none	none
X_copy_buffer	6	none	input parameter
X_free_network	5	none	none
X_free_io	5	none	none
X_free_user	5	none	none

Table 3.1 Executive Controller Functions (Contd.).

Module	Completed?	Is code the exact implementation of algorithm?
XB_boot XCB_control xcontrol_in xcontrol_out X_command list_status XC_next_query XC_query_dsm X_report_error	yes yes yes yes yes yes yes yes yes	yes yes yes yes yes yes yes yes yes
X_set_memory X_more_memory X_mem_error X_allocate X_reallocate X_free X_set_stack	yes yes yes yes yes yes yes	For none of the functions in this group, the algorithm is defined. For one or two line functions (which most of them are), it does not matter. But for the others at least comments in the code would help.
X_set_queue X_add_queue X_next_queue X_join_queue X_find_signature	yes yes yes yes yes	For none of the functions in this group, the algorithm is defined. For one or two line functions (which most of them are), it does not matter. But for the others at least comments in the code would help.
X_set_messages X_new_message X_free_message X_copy_message X_new_buffer X_free_buffer X_copy_buffer X_free_network X_free_io X_free_user	yes yes yes yes yes yes yes yes yes yes	For none of the functions in this group, the algorithm is defined. For one or two line functions (which most of them are), it does not matter. But for the others at least comments in the code would help.

Table 3.1 Executive Controller Functions (Contd.).

3.2 Executive Router Manager

Data Structures:

1. Consumer table: consists of,

    - consumer ids,
    - consumer names,
    - routing manager component specs which processes messages to/from consumer
    - consumer data specific to the routing manager component

2. Name table: consists of,

    - status,
    - executive id

3. Pending message queue

4. Routing component control structure: data to maintain and access executive routing components.

    - component ids,
    - component data,
    - entry points

5. Entry point table: executive routing manager entry point table.

Functions:

The functions below have been divided into 3 groups. The executive routing manager functions are defined in xrouter.c. These functions perform processing common to all routing algorithms. The operating system supports 2 routing algorithms: simple (standard routing algorithm) and NMR (N-Module Redundancy algorithm). The other 2 groups of functions in Table 3.2 contain functions used to implement them. They are defined in the files: xsimple.c and xnmr.c.

Module	Parameters	Module Description
XCB_router	data	router initialization entry point
XR_in	message	router in entry point
XR_out	message	router out entry point
XR_assign	data	router assign entry point
XR_control	code, data, router_id	router control entry point
new_name	id, state, consumer_id	allocates a name entry in the routing table.
new_consumer	id, router_id	allocates a consumer entry in the routing table.
lookup_name	reply	processes a lookup reply from DSM
X_route_junction	message, route_ptr, route_count	routes a message to a junction branch
X_route_consumer	message, consumer_id	send a message to a consumer
XRI_simple	router	simple router initialization entry pt.
xsimple_in	router, message, consumer	simple router in entry point
xsimple_out	router, message consumer	simple router out entry point
xsimple_assign	router, consumer	simple router assign entry point
xsimple_control	router, code, data	simple router control entry point
XRI_nmr	router	NMR router initialization entry point
xnmr_in	router, nmr, message	NMR router in entry point
xnmr_out	router, nmr, message	NMR router out entry point
xnmr_assign	router, consumer	NMR router assign entry point
xnmr_control	router, code, data	NMR router control entry point
xnmr_out_new	router, nmr, message	receives first message copy from consumer
xnmr_out_error	router, nmr, check, message	receives message copy before valid copy created
xnmr_out_valid	router, nmr, check, message	receives message copy after valid copy created
xnmr_ready	router, nmr, check, message	creates valid message
xnmr_done	router, nmr, check	message reception complete

Table 3.2 Executive Routing Manager Functions.

Module	Lines in code	Calls made to	Return Value
XCB_router	35	X_set_stack X_allocate init. entry pts. for routing algorithm	none
XR_in	12	exec. controller out & routing alg. in entry points	none
XR_out	44	X_report_error new_name X_add_queue X_next_query XC_query_dsm exec. manager out & routing algorithm in entry points	none
XR_assign	55	new_consumer new_name executive manager assign, routing algorithm assign & exec. router out entry points	none
XR_control	19	X_report_error routing algorithm control entry pts.	none
new_name	32	X_reallocate	ptr. to name data structure
new_consumer	26	X_reallocate	consumer data structure ptr
lookup_name	8	X_report_error XR_assign	none
X_route_junction	18	X_copy_message executive router in entry pts.	none
X_route_consumer	19	X_copy_message executive router out entry pts	none
XRI_simple	9	none	none
xsimple_in	7	X_route_consumer	none
xsimple_out	7	exec. res. manager out entry points	none
xsimple_assign	9	X_allocate	none
xsimple_control	13	X_report_error	none

Table 3.2 Executive Routing Manager Functions (Contd.).

Module	Lines in code	Calls made to	Return Value
XRI_nmr	13	X_set_stack X_allocate	none
xnmr_in	7	X_route_consumer	none
xnmr_out	14	xnmr_out_new xnmr_out_error xnmr_out_valid xnmr_done	none
xnmr_assign	11	none	none
xnmr_control	7	X_report_error	none
xnmr_out_new	28	X_add_queue xnmr_ready	NMR message structure ptr
xnmr_out_error	25	X_add_queue X_next_queue xnmr_ready	none
xnmr_out_valid	17	X_add_queue	none
xnmr_ready	16	X_copy_message xnmr_done exec. res. manager out entry pts.	none
xnmr_done	52	X_add_queue X_next_queue X_free_buffer X_free_message X_report_error	none

Table 3.2 Executive Routing Manager Functions (Contd.).

Module	Completed?	Is code the exact implementation of algorithm?
XCB_router	yes	yes
XR_in	yes	yes
XR_out	yes	yes
XR_assign	yes	X_add_queue & X_next_queue called in code, but not listed.
XR_control	yes	yes
new_name	yes	yes
new_consumer	yes	yes
lookup_name	yes	yes
X_route_junction	yes	yes
X_route_consumer	yes	yes
XRI_simple	yes	yes
xsimple_in	yes	yes
xsimple_out	yes	yes
xsimple_assign	yes	yes
xsimple_control	yes	yes, but no control functions implemented.
XRI_nmr	yes	yes
xnmr_in	yes	yes
xnmr_out	yes	yes
xnmr_assign	yes	yes
xnmr_control	yes	no, also no control functions implemented.
xnmr_out_new	yes	yes
xnmr_out_error	yes	yes
xnmr_out_valid	yes	yes
xnmr_ready	yes	yes
xnmr_done	yes	1/2 algorithm statements not in code

Table 3.2 Executive Routing Manager Functions (Contd.).

Note: Some of the executive routing functions in Tables 3.2 definitely need comments. For example function: XR_assign. It is extremely difficult to follow such functions as they have nested loops and 'if' statements.

3.3 Executive Resource Manager

Data Structures:

1. Resource table:

   - system and local resource ids,
   - manager component specs.

2. Resource manager component control structures: data to maintain and access resource manager components

   - component ids,
   - entry points,
   - component data

3. Executive ids to manager component and local ids mapping

4. Entry point table

Functions:

Table 3.3 contains the executive resource manager functions. The operating system currently supports 3 types of resources - network, I/O and application, and respective functions are grouped in the order described below. The executive manager functions perform processing common to all resources and are defined in the file xmanager.c, xnetwork.c contains network resource manager functions, xio.c has I/O resource manager functions and xprocess.c contains application resource manager functions.

Module	Parameters	Module Description
XCB_manager	data	manager initialization entry point
XM_in	xid, data	manager in entry point
XM_out	xid, data	manager out entry point
XM_assign	resource_id, name	manager assign entry point
XM_control	code, xid, data	manager control entry point
xm_enter	name	enters an executive id in manager tables
XMI_network	manager	n/w manager initialization entry point
xnetwork_in	manager, packet local_id	n/w manager interrupt entry point
xnetwork_out	manager, message path_id	n/w manager output entry point
xnetwork_assign	manager, name, resource_id	n/w manager assign resource entry point
xnetwork_control	manager, code, local_id, data	n/w manager control function entry point
XMI_io	manager	I/O manager initialization entry point
xio_in	manager, packet local_id	I/O manager interrupt entry point
xio_out	manager, message local_id	I/O manager output entry point
xio_assign	manager, name, resource_id	I/O manager assign resource entry point
xio_control	manager, code, local_id, data	I/O manager control function entry point
xio_send	manager, io, message	processes output data message
xio_query	manager, io, message	processes output query message
xi_reset_junction	manager, local_id, junction	reset I/O resource consumer junction

Table 3.3 Executive Resource Manager Functions.

Module	Parameters	Module Description
XMI_process	manager	application manager init. entry point
xprocess_in	manager, packet local_id	application manager interrupt entry pt.
xprocess_out	manager, message local_id	application manager output entry pt.
xprocess_assign	manager, name, resource_id	application manager assign resource entry point
xprocess_control	manager, code, local_id, data	application manager control function entry point
process_accept	manager, packet, process	processes application accept packet
process_query	manager, packet, process	processes application query packet
process_reply	manager, packet, process	processes application reply packet
process_call	manager, packet, process	processes application call packet
process_receive	manager, packet, process	processes application receive packet
process_send	manager, packet, process	processes application send packet
process_ready	manager, packet, process	checks for application process ready packet
match_query	manager, packet, process	checks query packet
process_out_query	manager, process, message	processes output query message
process_out_reply	manager, process, message	processes output reply message
process_out_send	manager, process, message	processes output send message
process_run2wait	manager, process	disable application consumer scheduling
process_wait2ready	manager, process	enable application consumer scheduling
next_process	manager	schedule next application consumer
xp_reset_junction	manager, junction, local_id	reset application consumer junction

Table 3.3 Executive Resource Manager Functions (Contd.).

Module	Lines in code	Calls made to	Return Value
XCB_manager	76	X_allocate, routing alg. init. entry pts.	none
XM_in	13	resource manager in & exec. control in entry points	none
XM_out	13	resource manager out & exec. control out entry points	none
XM_assign	20	X_report_error resource manager assign	none
entry points			
XM_control	23	xm_enter X_report_error resource manager control entry points	none
xm_enter	28	X_reallocate	none
XMI_network	52	X_allocate, exec. manager control & kernel assign entry pts.	none
xnetwork_in	13	X_new_message X_free_network exec. router out entry pts	none
xnetwork_out	17	X_free_buffer X_free_message kernel out entry pts	none
xnetwork_assign	25	X_report_error exec. manager control entry points	none
xnetwork_control	8	X_report_error	none
XMI_io	37	X_reallocate	none
xio_in	36	X_next_queue X_free_io X_new_message X_add_queue X_route_junction exec. router in entry pt.	none
xio_out	14	xio_send xio_query X_report_error	none
xio_assign	44	X_allocate X_reallocate X_report_error exec. manager control & kernel link assign entry pt.	none

Table 3.3 Executive Resource Manager Functions (Contd.).

Module	Lines in code	Calls made to	Return Value
xio_control	11	xi_reset_junction X_report_error	none
xio_send	14	X_free_buffer X_free_message kernel out entry pts.	none
xio_query	19	X_next_queue X_free_io X_add_queue exec. router in entry pt.	none
xi_reset_junction	11	X_reallocate	none
XMI_process	41	X_allocate	none
xprocess_in	21	process_accept process_query process_reply process_call process_receive process_send process_ready X_report_error	none
xprocess_out	16	process_out_query process_out_reply process_out_send X_report_error	none
xprocess_assign	52	X_allocate X_reallocate X_report_error process_wait2ready exec. manager control & kernel link assign entry pts	none
xprocess_control	11	xp_reset_junction X_report_error	none

Table 3.3 Executive Resource Manager Functions (Contd.).

Module	Lines in code	Calls made to	Return Value
process_accept	20	X_next_queue X_free_message X_add_queue kernel out entry pt.	none
process_query	10	match_query kernel out entry pt.	none
process_reply	17	X_new_message X_free_user executive router in & kernel out entry pts.	none
process_call	8	match_query process_ready	none
process_receive	20	X_next_queue X_free_message X_add_queue process_ready	none
process_send	37	X_new_message X_route_junction X_report_error X_free_user executive router in & kernel out entry points	none
process_ready	16	X_next_queue X_free_user X_free_buffer process_run2wait kernel out entry points	none
match_query	52	X_find_signature X_add_queue X_new_message X_free_message X_route_junction X_report_error exec. router in entry pt.	none

Table 3.3 Executive Resource Manager Functions (Contd.)

Module	Lines in code	Calls made to	Return Value
process_out_query	20	X_free_message X_add_queue process_wait2ready	none
process_out_reply	22	X_find_signature X_add_queue X_free_message process_wait2ready	none
process_out_send	22	X_add_queue X_free_message process_wait2ready	none
process_run2wait	7	next_process	none
process_wait2ready	8	next_process X_add_queue	none
next_process	17	X_next_queue X_free_user X_free_buffer kernel out entry point	none
xp_reset_junction	12	X_reallocate	none

Table 3.3 Executive Resource Manager Functions (Contd.).

Module	Completed?	Is code the exact implementation of algorithm?
XCB_manager	yes	code long, hard to follow without comments
XM_in	yes	yes
XM_out	yes	yes
XM_assign	yes	yes
XM_control	yes	yes
xm_enter	yes	yes
XMI_network	yes	code long, hard to follow without comments
xnetwork_in	yes	yes
xnetwork_out	yes	yes
xnetwork_assign	yes	yes
xnetwork_control	yes	no, also no control functions implemented.
XMI_io	yes	yes
xio_in	yes	code long, hard to follow without comments
xio_out	yes	yes
xio_assign	yes	code long, hard to follow without comments
xio_control	yes	yes
xio_send	yes	yes
xio_query	yes	yes
xi_reset_junction	yes	yes
XMI_process	yes	code long, hard to follow without comments
xprocess_in	yes	yes
xprocess_out	yes	yes
xprocess_assign	yes	code long, hard to follow without comments
xprocess_control	yes	yes
process_accept	yes	yes
process_query	yes	yes
process_reply	yes	yes
process_call	yes	yes
process_receive	yes	yes
process_send	yes	yes
process_ready	yes	yes
match_query	yes	code long, hard to follow without comments
process_out_query	yes	yes
process_out_reply	yes	yes
process_out_send	yes	yes
process_run2wait	yes	yes
process_wait2ready	yes	yes
next_process	yes	yes
xp_reset_junction	yes	yes

Table 3.3 Executive Resource Manager Functions (Contd.).

4. DISTRIBUTED SYSTEM MANAGER DATA STRUCTURES & FUNCTIONS

The distributed system manager (DSM) monitors and maintains the state of the entire distributed system. It provides a high level fault management, assigns and activates resources to implement consumers and monitors the overall resource performance. DSM has four components: the DSM controller, DSM resource manager, DSM scheduler and DSM fault manager. The modules and data structures in these components are described below.

4.1 DSM Controller

Data Structures:

Currently, the actual data structures to support the controller activity are not well defined.

Functions:

The DSM controller functions are given in Table 4.1. The function "main" is defined in the file dsm.c. The file mboot.c contains the DSM boot functions. Finally, the distributed system manager commands are in mcommand.c. Table 4.1 has grouped functions in the same files.

Module	Parameters	Module Description
main	none	DSM mainline function.
mc_boot	system, command	boot entry point for DSM
mc_add_manager	manager_ptr, system	adds new manager entry to manager list
mc_add_router	router_ptr, system	adds new router entry to router list
mc_add_resource	resource_ptr, system	adds new resource entry to resource list
mc_add_exec	exec_ptr, system	adds new executive entry to executive list
mc_add_link	system, link, exec_id	adds new link entry to executive link list
network_link	system, link, exec_id	adds n/w link entry to executive link list
io_link	system, link, exec_id	adds io link entry to executive link list
appl_link	system, link, exec_id	adds process link entry to executive link list
mgr_command	command	executes a DSM command
mc_list	system, command	executes a list of DSM commands of given length
mc_exec_error	system, command	prints out an error message
mc_undefined	system, command	prints out an undefined command error msg.

Table 4.1 DSM Controller functions

Module	Lines in code	Calls made to	Return Value
main	11	sys_accept sys_read mgr_command	none
mc_boot	81	U_set_memory U_set_stack U_set_command U_allocate mc_add_manager mc_add_router mc_add_resource mc_add_exec mc_add_link enter_consumer enter_name assign_resource	none
mc_add_manager	13	U_allocate	none
mc_add_router	16	U_allocate	none
mc_add_resource	35	U_allocate	none
mc_add_exec	29	U_allocate enter_consumer enter_name	none
mc_add_link	13	network_link io_link appl_link	none
network_link	20	none	none
io_link	13	none	none
appl_link	13	none	none
mgr_command	6	mc_list, mc_boot mc_unknown_con mc_unknown_name mc_define mc_link, mc_run mc_get_consumer mc_get_cpu mc_exec_error mc_undefined	none
mc_list	14	same as mgr_command	none
mc_exec_error	8	none	none
mc_undefined	6	none	none

Table 4.1 DSM Controller functions (Contd.)

Module	Completed?	Is code the exact implementation of algorithm?
main	yes	yes
mc_boot	yes	code long, hard to follow without comments
mc_add_manager	yes	yes
mc_add_router	yes	yes
mc_add_resource	yes	yes
mc_add_exec	yes	yes
mc_add_link	yes	slight discrepancy between code and algorithm
network_link	yes	yes
io_link	yes	yes
appl_link	yes	yes
mgr_command	yes	yes
mc_list	yes	yes
mc_exec_error	yes	yes
mc_undefined	yes	yes

Table 4.1 DSM Controller functions (Contd.)

4.2 DSM Scheduler

Data Structures:

Currently, the actual data structures to support the scheduler activity are not well defined.

Functions:

The DSM scheduler functions are defined in the following files: mcdefine.c, mclink.c, mcrun.c, mcstatus.c and mcunknown.c. The functions in the tables below are grouped accordingly.

Module	Parameters	Module Description
mc_define assign_resource assign_io assign_appl assign_network sort_execs sort_network	system, command system, consumer, manager_id, mask system, mask, consumer system, mask, consumer system, exec_id, consumer system, exec_id, system, manager,	defines a consumer. assigns exec. with available resources to a consumer assigns executives with available io resources to a consumer assigns executives with available process resources to a consumer assigns network between consumer executives sorts system executive table sorts networks in system resource table
mc_link	system, command	defines a link between consumers.
mc_run run_tos	system, command system, consumer	begins execution of a consumer adds consumer address to all execs. which must communicate with it
mc_get_consumer mc_get_cpu	system, command system, command	sends ack. with status info. describing the given consumer sends ack. with status info. describing the given executive
mc_unknown_con mc_unknown_name	system, command system, command	adds consumer address to an executive which must communicate with it module not implemented

Table 4.2 DSM Scheduler Functions.

Module	Lines in code	Calls made to	Return Value
mc_define	48	r_lookup rtr_lookup enter_consumer enter_name assign_resource assign_network	none
assign_resource	25	assign_io assign_appl	# of executives assigned to the consumer
assign_io	40	sort_execs	# of executives assigned to the consumer
assign_appl	40	sort_execs	# of executives assigned to the consumer
assign_network	70	sort_networks	# of additional networks assigned to the consumer
sort_execs	29	none	none
sort_network	34	none	none
mc_link	60	c_lookup set_junction reset_junction U_allocate U_free	none
mc_run	23	c_lookup U_allocate run_tos run_names	none
run_tos	46	lookup_link next_x_command issue_x_command	none
mc_get_consumer	64	c_lookup next_ack, U_copy	none
mc_get_cpu	19	x_lookup sys_get_message sys_send_query cpu_report next_ack U_pop_stack	none
mc_unknown_con	29	assign_network next_ack lookup_link	none
mc_unknown_name	6	none	none

Table 4.2 DSM Scheduler Functions (Contd.)

Module	Completed?	Is code the exact implementation of algorithm?
mc_define	yes	yes
assign_resource	yes	algorithm does not quite explain the code
assign_io	yes	yes
assign_appl	yes	algorithm does not quite explain the code
assign_network	yes	code too long, slightly difficult to follow, needs comments
sort_execs	yes	yes
sort_network	yes	yes
mc_link	yes	code too long, slightly difficult to follow, needs comments
mc_run	yes	yes
run_tos	yes	yes
mc_get_consumer	yes	yes
mc_get_cpu	yes	yes
mc_unknown_con	yes	yes
mc_unknown_name	no	this module has not been implemented

Table 4.2 DSM Scheduler Functions (Contd.)

4.3 DSM Resource Manager

Currently, the DSM resource manager modules are implemented as local functions within the DSM controller and scheduler components.

4.4 DSM Fault Manager

No documentation about modules or data structures are given in the FTDCS OS manuals.

APPENDIX C

STRUCTURE DESIGN DOCUMENT
FOR THE
FTDCS SIMULATOR & THE OPERATING SYSTEM

Table of Contents

List of Figures

page

1. INTRODUCTION

This appendix describes in detail the Structure Design for both the FTDCS simulator and operating system. A hierarchical approach is taken in structure design. It begins with an overview of the general design, followed by a detailed decomposition.

Section 2 explains the structure design diagram conventions.

Sections 3 and 4 explain in detail the structure design of the FTDCS simulator and operating system respectively. The simulator and the operating system (e.g., kernel) descriptions are based on the system development example explained in Chapter 7 of "FTDCS Software Development: System Programmer's Guide".

The structure diagrams are given at the end of Section 4.3. The connectors used throughout the structure diagrams are listed for easy reference in Appendix C1.

2. STRUCTURE DESIGN

2.1 Purpose of Structure Design

The structure design defines the physical specifications to accomplish the data transformations. This design corresponds to the physical system and shows the processing and control through structure diagrams.

2.2 Structure Design Diagram Conventions

Structure diagrams in this document use the following symbol conventions:

Rectangles and squares correspond to processes and are designated with a task name. Each diagram contains a process tree consisting of a parent process and the children processes that it calls. A particular parent process may also appear as a child on other structure diagrams.

Invocation arrows are used to show that control is passed from a parent process to a child process, and also passed back again when the called process has finished execution. There are two ways in which a child process may be called by a parent process: either directly or upon user input. The former case is shown with a solid line having an arrow at its end between the processes, while a dotted line with an arrow at its end is used to depict the latter. Processes at the called end of a dotted line correspond to processes invoked upon specific user input (often options). In Figure EX1 for example, the process B is called directly by the parent process A, and control returns to A when B is finished. Process A simply enables process C, but does not invoke it; it will be executed upon specific user input corresponding to process C. Upon completion of C, control returns to A, but C is still enabled.

The sequence with which a set of child processes are called by a common parent process is shown by a direction arrow through the invocation arrows. The children of process B in Figure EX1 for example, is called in the order D, E and F.

Conditional process control flow is designated by a diamond-shaped condition symbol that is attached to a process. In Figure EX1, for example, process D can call either G or H. Condition symbols occur in two forms: automatic or user option. User instigated options have dotted lines and processes are selected as a function of an external choice. Automatic selection occurs as a result of a condition being met by the system. Such connections can be thought of as traditional "if .... then ..." condition links.

Looping is shown by a rounded arrow through the invocation arrows of the processes called in the loop. In Figure EX1, process F loops through calls to processes I and J.

Certain processes may be invoked at different times by several other processes. Moreover two processes drawn at the two ends of a structure diagram may have the same process as their child process. Drawing links from these processes to the child process may make the diagram messy and difficult to understand. Therefore in such cases, instead of redrawing the process every time it is invoked, connectors are used. The very first time a process is described, it is drawn as a rectangle or a square. For all subsequent occurrences of that process, its connector is drawn. A connector is a circle with the process name (full or abbreviated) and figure number (corresponding to the figure where it is described as a regular process) in it. For example, "X" is a connector in Figure EX1. For the structure diagrams in this document, a list of connectors, (with the corresponding process's full name, the process identification number and a list of all the figures in which this connector appears) is given in Appendix C1.

The names for some processes describe the process functionality in brief (e.g., "display error msg." displays an error message). However, since the FTDCS structure design has been developed from the existing code, majority of the processes are named after the corresponding "C" functions (e.g., "X_report_error"). This makes it easier to compare the two.

The following numbering convention is used for the processes. Each child process is related back to the parent process through its identification number designation, which is a decimal of the parent process. For example, process 2 has three children processes 2.1, 2.2 and 2.3; 2.1 is decomposed to 2.1.1, 2.1.2 etc., and so on (see Figure EX1).

In the following sections, the main process and its subprocesses are explained in a breadth-first manner (i.e., in the sequence 1,2, ... n; 1.1, 1.2 ... 1.n; 2.1, 2.2 ... 2.n etc.). The numbers correspond to the process identification numbers shown on the figures. Processes which do not contain subprocesses are indicated as such with a hollow circle appearing at the end of their descriptions.

The following two sections explain the structure design processes for the FTDCS simulator and the operating system in detail.

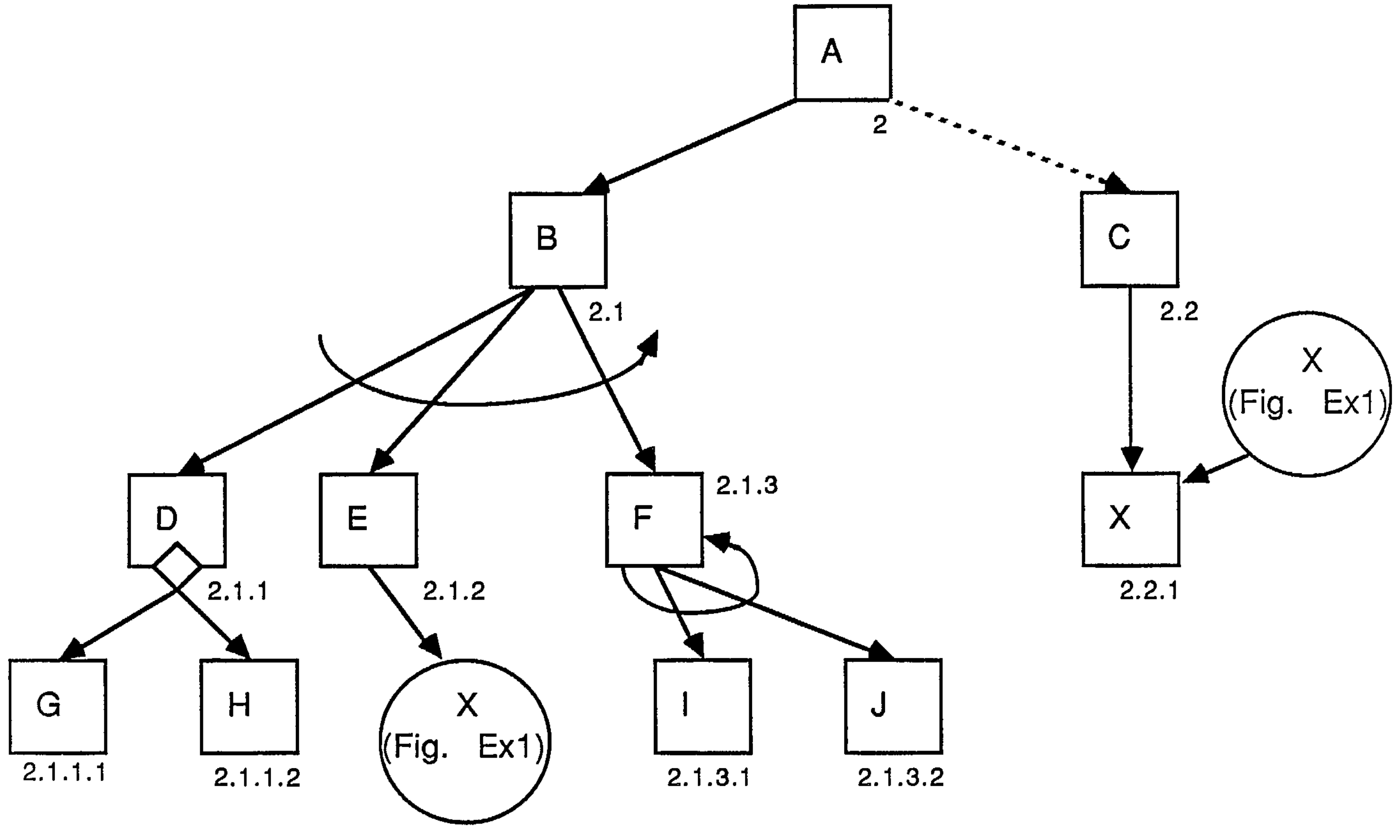


Figure EX1: Example Structure Diagram

3. SIMULATOR STRUCTURE DESIGN

The structure design diagrams and descriptions for the FTDCS simulator are given below. The corresponding structure diagrams are shown in Figures S1 through S19.

Main (Simulator)

This is the mainline process for the simulator. It controls the entire simulation. First it calls "st_system" to define the system model.

Next, it makes available a set of commands for the user, and waits for the user to select a command. When the user selects a command, it calls the respective process to execute that command. For example, it calls "st_memory" if the user wants a display of the memory status. Typing the wrong command by the user will make the system display a list of choices ("show command list"). Execution of commands can be repeated as many times as necessary, until the user decides to quit. In order to terminate the simulator, the user must type 'x'. This will cause the process to make calls to other processes to free the simulator, stop the CPU and delete the system model. See Figure S1.

1. st_system

This function creates a simulator system definition model from the hardware definition and the distributed software specifications. It calls "st_set_system" to initialize the system definition structure, "st_sw_config" to read and interpret the distributed software specifications for the model and "st_sys_config" to interpret the hardware definition for the model. See Figures S1 and S2.

2. st_memory

This process displays the memory status. It calls "get memory status" to get the memory information and displays this information to the user by a call to "show memory status". See Figures S1 and S3.

3. st_io

This process simulates an I/O event. Initially, it gets the resource type input by the user. If the resource is not an I/O device, an error message is displayed ("display error message"). Otherwise, "sim_to_resource" is called to simulate an I/O event. See Figures S1 and S4.

4. st_console

This process simulates an I/O console. Initially, it gets the resource type input by the user. If the resource is not an I/O device, an error message is displayed ("display error message"). Otherwise, "sim_to_resource" is called repeatedly to simulate an I/O console. See Figures S1 and S5.

5. st_file

This process creates a configuration file for the CPU. It calls "st_get_cpu" to get the processor for configuration from the user. Next, the processor configuration data structure is created ("st_config"). Also, configuration data and functional configuration table files are created (calls to "stf_data" and "stf_tables"). Finally, the configuration data structure is deallocated. See Figures S1 and S6.

6. st_go

This process starts the simulation. For every defined processor, this process calls "st_config" to create a local operating system configuration structure based on the system definition and the local configuration specification, "sim_cpu_go" to start the simulation and lastly, "st_free_config" to free the configuration data structures. Finally "cpu_run" is called to include the event in the kernel queue for execution. See Figures S1 and S7.

7. st_node

This process configures a node for testing the system. It gets the CPU name from the user and looks up the CPU id. If the CPU id is empty, a message is displayed stating that the processor is not defined. Otherwise, a local operating system configuration structure is created based on the system definition and the local configuration specification ("st_config"), simulation is started ("sim_cpu_go"), and the configuration data structures are freed ("st_free_config"). Finally, "cpu_run" is called to include the event in the kernel event queue for execution. See Figures S1 and S8.

8. sm_show

This process shows the hardware configuration. See Figure S1. ○

9. st_disable

This process disables a component. See Figure S1. ○

10. st_enable

This process enables a component. See Figure S1. ○

11. show command list

This process displays a list of simulator commands, what they do and how they can be invoked. See Figure S1. ○

12. sim_cpu_stop

This process frees the memory allocated for the operating system structures and the CPU's OS memory. See Figure S1. ○

13. st_free_system

This process deletes a system model definition, freeing its allocated memory. See Figure S1. ○

1.1 st_set_system

This function initializes the system definition structure. The ids are set for each resource manager ("set resource manager ids") and each resource ("set resource ids"). Finally, "set processor links" sets ids and creates a table of resource links for each processor. See Figures S2 and S9.

1.2 st_sw_config

This process reads and interprets the distributed software specification for the system definition model. Initially, it gets the distributed software specification file name from the user and accesses the data in it. Next, the DSM consumer definition command list is initialized, memory is allocated for the processor table and each entry in the table is set to empty. The DSM name count and the processor mask are both set to 0. The DSM consumer id is set to the number of system processors.

Next, this process calls other processes to read and set DSM resource, router and processors. Finally "read consumers" is called to read a list of application consumer specifications and create DSM commands to implement them. See Figures S2 and S10.

1.3 st_sys_config

This process interprets the hardware definition and integrates it to the system definition model. It initializes the system definition structure. Space is allocated for the system definition header. Next, the system DSM data is set. So are the resource manager count, routing manager count, resource count, processor count and processor/link count. The initial DSM table sizes are also set.

Next, each resource manager is added to the system definition ("add_sys_manager") as are the routing managers ("add_sys_router"), resources ("add_sys_resource") and processors ("add_sys_exec"). See Figures S2 and S11.

2.1 get memory status

This process retrieves the memory status information (such as the number of used and free, bytes and blocks of memory). See Figure S3. ○

2.2 show memory status

This process displays the memory status information to the user (which includes information such as the number of used and free, bytes and blocks of memory). See Figure S3. ○

3.1 display error message

This process displays the error message (given to it as its input) to the user (e.g., resource not IO message, etc.). See Figure S4. ○

3.2 sim_to_resource

This process simulates an I/O console or an I/O event. It checks to see if the given resource has any assigned links. If the resource does not have any assigned links, an error message is displayed.

Otherwise, the process gets the system model id and thus the model simulator id. A message is created ("new_message") and set to the simulator message. Finally, the message is included in the kernel event queue for execution ("cpu_run"). See Figures S4 and S12.

5.1 st_get_cpu

This process gets the processor for which the configuration data and functional configuration files are to be created, from the user. See Figure S6. ○

5.2 st_config

This process creates a local operating system configuration structure based on the interpretation of the system definition and the local configuration specification. This structure is used to create the functional configuration table file. It initializes configuration data management and processor configuration pointer. Next, the configuration data header is setup ("st_config_header"). Finally, the processor (an Intel 8086) dependent portion of the configuration data structure is created ("st86_config"). See Figures S6 and S13.

5.3 stf_data

This process produces the configuration data file from the configuration data for a processor. First, it attaches the prefix "cfg" and suffix ".c" to the file name. The process then tries to create and open the configuration data file. If the file cannot be created, an error message is displayed.

If the file is created successfully, the following information is written to the file. The configuration data header ("add_header"), kernel configuration data ("add_kernel"), executive configuration data ("add_exec") and the DSM configuration data ("add_dsm"). The file is then closed. See Figures S6 and S14.

5.4 stf_tables

This process produces the functional configuration table file from the configuration data for a processor. Initially it attaches the prefix "tbl" and suffix ".c" to the file name. The process then tries to create and open the functional configuration table file. If the file cannot be created, an error message is displayed.

If the file is created successfully, the following information is written to the file. Configuration table headers ("tbl_header"), handler initialization entry points ("tbl_handlers") and server initialization entry points ("tbl_servers"). The file is then closed. See Figures S6 and S15.

6.1 sim_cpu_go

This process initializes the operating system structures with the required kernel, executive and DSM initialization routines. Next, it calls the kernel boot process ("KB_boot").

See Figures S7 and K1. Note: this process is the same as process 1 in Figure K1.

6.2 st_free_config

This process frees the data structures allocated for the configuration data. See Figure S7. ○

6.3 cpu_run

This process queues each message at the tail of the kernel event queue for future execution. It gets the CPU id for the system model and thus the operating system data for the CPU. Next, it calls "K_cpu_enter" to make the context switch to the OS context. "K_cpu_fork" is called after this to add the event to the kernel event pending queue. Finally the system is returned to normal operation ("K_cpu_exit"). See Figures S7 and K2.

Note: this process calls the OS kernel functions, which are described in Section 4.1 below. This process is the same as the process 2 in Figure K2.

1.1.1 set resource manager ids

System identifiers in consecutive order are assigned to each resource manager in the system definition model by this process. See Figure S9. ○

1.1.2 set resource ids

System identifiers in consecutive order are assigned to each resource in the system definition model by this process. See Figure S9. ○

1.1.3 set processor links

System identifiers in consecutive order are assigned to each processor in the system definition model by this process. Also, for each processor a table of its resource links is created. This table is sorted by resource manager. See Figure S9. ○

1.2.1 read & set DSM resource

This process reads the DSM resource name from the file and sets the DSM resource id to the system resource id. See Figure S10. ○

1.2.2 read & set DSM router

This process reads the DSM router name from the file and sets the DSM router id to the system router id. See Figure S10. ○

1.2.3 read & add DSM processors

This process reads processor names from the file and looks up the system id for processors. It then adds the processor id to both the DSM processor table and the DSM processor mask. Also for each processor, the name count is incremented. See Figure S10. ○

1.2.4 read consumers

This process reads a list of application consumer specifications and creates DSM commands to implement them. For each command read from the file, it calls the respective process (e.g., "std_define", "std_link" and "std_run" for the define, link and run commands respectively). For any other command, an error message is displayed. See Figure S10.

1.3.1 add_sys_manager

This process allocates space for the resource manager structure and the resource manager system id is set from the model resource manager. See Figure S11. ○

1.3.2 add_sys_router

This process allocates space for the routing manager structure and the routing manager system id is set from the model routing manager. Also, the routing manager name is copied from the model manager. See Figure S11. ○

1.3.3 add_sys_resource

This process allocates space for the resource structure. The resource system id, type, name, manager id and the link id are set from the model resource. See Figure S11. ○

1.3.4 add_sys_exec

This process allocates space for the executive structure. The executive system id and type are set from the model processor. The executive name and the link count are set from the model executive.

For each processor/resource link, space is allocated for the executive link structure. The link resource system id and the link unit id are set from the model. See Figure S11. ○

3.2.1 new_message

This process creates a message structure, sets its id, the cpu id, unit id, time lag, length, and the message data (which includes the header and the actual data) and returns the new message. See Figure S12. ○

5.2.1 st_config_header

This process sets up the header for a local executive configuration data structure. The configuration data length and the executive id are set. The lengths of the kernel, executive and DSM configuration data are set to 0. See Figure S13. ○

5.2.2 st86_config

This process creates configuration data structure (the processor dependent and independent portions) for an Intel 8086 processor's local operating system running on the simulator. The local configuration specification file is read by a call to "st86_read". Appropriate processes are called to initialize the following structures:
- executive resource manager ("st86_managers"),
- handler configuration data ("st86_handlers"),
- server configuration data ("st86_servers"),
- resource configuration data ("st86_resources"), and
- processor/resource links configuration data ("st86_links").

The kernel ("st86_kernel") and processor independent configuration data ("st_config_exec") are also added. See Figures S13 and S16.

5.3.1 add_header

This process writes the configuration data header to the configuration data file. The following information is written to the file:
- configuration file title,
- configuration code,
- processor system id,
- configuration data length for kernel, executive and DSM and the total configuration data length. See Figure S14. ○

5.3.2 add_kernel

This process writes the kernel configuration data to the configuration data file. The following information is written to the file: kernel memory manager configuration data ("add_kmemory"), kernel processor manager configuration data ("add_kcpu") and kernel link manager configuration data ("add_klink"). See Figure S14.

5.3.3 add_exec

This process writes the executive configuration data to the configuration data file. The following information is written to the file:
- executive controller, resource and routing manager configuration data,
- the resource and link data id for each resource linked to the processor,
- executive consumer name and data for each executive linked to the processor,
- DSM consumer data,
- DSM consumer name data for each DSM consumer name,
- local DSM consumer name and junction, if DSM is scheduled on a processor. See Figure S14. ○

5.3.4 add_dsm

This process writes the DSM configuration data to the configuration data file. It writes a header to the file and calls "add_dsm_command" to write DSM list command to the file. See Figure S14.

5.4.1 tbl_header

This process writes the configuration table header to the functional configuration table file. The following information is written to the file:
- title of the file,
- C include statements for machine data types, processor independent table data and configuration data file.

See Figure S15. ○

5.4.2 tbl_handlers

This process writes the handler initialization entry points to the functional configuration table file. The following information is written to the file:
- external function declaration for each handler in configuration data,
- handler initialization entry point for each handler in configuration data. See Figure S15. ○

5.4.3 tbl_servers

This process writes the server initialization entry points to the functional configuration table file. The following information is written to the file:
- external function declaration for each server in configuration data,
- server initialization entry point for each server in configuration data. See Figure S15. ○

1.2.4.1 std_define

This process reads the consumer's name, its resource and router names and its name count from the input file. It then creates a define consumer command in the command buffer. See Figure S10. ○

1.2.4.2 std_link

This process reads the consumer's name from the input file and initializes a link consumer command. It then reads the consumer name for link and the branch flag from the input file and adds link to the input consumer. See Figure S10. ○

1.2.4.3 std_run

This process reads the consumer's name from the input file and creates a run consumer command in the command buffer. See Figure S10. ○

5.2.2.1 st86_managers

This process initializes the configuration data structures for the executive resource managers. It sets the manager count to the number of simulator resource managers and allocates memory for resource manager data structures. The manager id, server and link counts of each resource manager data structure are initialized. See Figure S16. ○

5.2.2.2 st86_handlers

This process initializes the configuration data structures for the kernel interrupt handlers. It sets the handler count to 0 and allocates memory for handler data structures. The handler link count of each interrupt handler is set to 0. See Figure S16. ○

5.2.2.3 st86_servers

This process initializes the configuration data structures for the kernel link servers. It sets the server count to 0 and allocates memory for server data structures. The server link count of each server is set to 0. See Figure S16. ○

5.2.2.4 st86_resources

This process initializes the configuration data structures for the processor's linked resources. The resource count is set to 0, memory is allocated for resource data structure and the resource data structures are set to empty. See Figure S16. ○

5.2.2.5 st86_read

This process reads a local configuration specification file for an 8086 family processor. For each configuration line read from the file, it either calls "add_handler" (to add handler data to configuration data), "add_server" (to add server data to configuration data), "add_resource" (to add resource data to configuration data) or "display error message" (for an unknown command). See Figures S16 and S17.

5.2.2.6 st86_links

This process initializes the configuration data structures for the processor's resource links. The link count is set to the number of processor links and link data structures to empty. Each processor link's handler link count, resource manager link count and server link count are incremented. For each kernel interrupt handler, the handler unit base and configuration unit count are modified. See Figure S16. ○

5.2.2.7 st86_kernel

This process adds the kernel configuration data to the configuration data structures. In order to do this, it calls processes to add the kernel memory manager ("st86_kmemory"), kernel processor manager ("st86_kcpu") and kernel link manager ("st86_klink") configuration data. See Figures S16 and S18.

5.2.2.8 st_config_exec

This process creates the executive and DSM portions of the local configuration data structure. Initially, it sets the executive and the DSM table and memory sizes, ids (e.g., executive id), counts (e.g., resource, routing manager, etc.).

It then calls "config_resources" to add resource data to executive configuration, "config_execs" to add linked executive data to executive configuration and "config_dsm" to add DSM configuration data to local configuration. See Figures S16 and S19.

5.3.2.1 add_kmemory

This process writes the kernel memory manager configuration data to the configuration data file. See Figure S14. ○

5.3.2.2 add_kcpu

This process writes the kernel processor manager configuration data to the configuration data file. See Figure S14. ○

5.3.2.3 add_klink

This process writes the kernel link manager configuration data to the configuration data file. See Figure S14. ○

5.3.4.1 add_dsm_command

This process writes DSM commands to the configuration data file. It writes a DSM command header to the output file. Next, depending upon the type of command, it calls the appropriate process which writes the command to the file (e.g., "add_dsm_boot" is called to write the boot command to the output file, "add_dsm_link" for the link command, etc.). See Figure S14.

5.2.2.5.1 add_handler

This process reads a handler definition from the local configuration specification file and adds it to the configuration data structure. See Figure S17. ○

5.2.2.5.2 add_server

This process reads a server definition from the local configuration specification file and adds it to the configuration data structure. See Figure S17. ○

5.2.2.5.3 add_resource

This process reads a resource definition from the local configuration specification file and adds it to the configuration data structure. Depending upon the type of the resource manager (e.g., network), the appropriate process is called to read the resource specific data (e.g., "add86_network"). See Figure S17.

5.2.2.7.1 st86_kmemory

This process adds the kernel memory manager configuration data structure to the configuration data structure. It allocates memory for the data structure. It also sets kernel memory manager data size, processor id, initial memory size and buffer management parameters. See Figure S18. ○

5.2.2.7.2 st86_kcpu

This process adds the kernel processor manager configuration data structure to the configuration data structure. It allocates memory for the data structure. It also sets kernel processor manager data size, kernel and executive event queue sizes. See Figure S18. ○

5.2.2.7.3 st86_klink

This process adds the kernel link manager configuration data structure to the configuration data structure. It allocates memory for the data structure. Next, it sets the kernel link manager data size, kernel id table parameters, interrupt handler count, unit count, server count, link count and link data size.

For each interrupt handler, space is allocated for the structure, and the handler data size, unit count and vector are set. For each link server, space is allocated for the structure, and server link count and data size are set.

For each processor resource link, depending upon the type of the link resource manager, the appropriate link (network, I/O, application) configuration data is added to the file ("st86_network_link", "st86_io_link" and "st86_process_link"). See Figure S18.

5.2.2.8.1 config_resources

This process adds executive resource data to the local configuration data for all resources which are available to a given executive. See Figure S19. ○

5.2.2.8.2 config_execs

This process adds a linked executive data structure to the executive portion of the local configuration data structure. Also, for each model processor, its executive consumer definition is also added ("add_exec_consumer"). See Figure S19.

5.2.2.8.3 config_dsm

This process adds the distributed system manager consumer to the local executive configuration data structure. If the DSM is assigned to the executive, then the DSM configuration data is also added to the local configuration data. See Figure S19. ○

5.3.4.1.1 add_dsm_boot

This process writes the parameters associated with a DSM boot command to the configuration data file. See Figure S14. ○

5.3.4.1.2 add_dsm_list

For every command in the list, this process calls "add_dsm_command" to add each command to the configuration data file. See Figure S14.

5.3.4.1.3 add_dsm_define

This process writes the parameters associated with a DSM define consumer command to the configuration data file. See Figure S14. ○

5.3.4.1.4 add_dsm_link

This process writes the parameters associated with a DSM link consumer command to the configuration data file. See Figure S14. ○

5.3.4.1.5 add_dsm_run

This process writes the parameters associated with a DSM run consumer command to the configuration data file. See Figure S14. ○

5.2.2.5.3.1 add86_network

This process reads the network resource specific data from the local configuration specification file and adds it to the configuration data structure. The data includes shared memory segment and shared memory offset used to synchronize communication across the multibus. See Figure S17. ○

5.2.2.5.3.2 add86_io

This process reads the I/O resource specific data from the local configuration specification file and adds it to the configuration data structure. The data includes shared memory segment and shared memory offset used to synchronize communication with the 188/48 communicating SBC. See Figure S17. ○

5.2.2.5.3.3 add86_process

This process reads the application process resource specific data from the local configuration specification file and adds it to the configuration data structure. This data consists of the load addresses and segment lengths of the process' object code. See Figure S17. ○

5.2.2.7.3.1 st86_network_link

This process adds the configuration data for a link to a network resource to the configuration data structure. It allocates space for the structure, and sets the link server, unit id, data size, resource id, address count, network link segment, offset and home address. See Figure S18. ○

5.2.2.7.3.2 st86_io_link

This process adds the configuration data for a link to an I/O resource to the configuration data structure. It allocates space for the structure, and sets the link server, unit id, data size and I/O link segment and offset. See Figure S18. ○

5.2.2.7.3.3 st86_process_link

This process adds the configuration data for a link to an application process resource to the configuration data structure. It allocates space for the structure, and sets the link server, unit id, data size and resource id. See Figure S18. ○

5.2.2.8.2.1 add_exec_consumer

This process adds an executive consumer definition to the local executive configuration data structure. The consumer id is set to the system id, router to simple, name count to 1, name id to exec. id. The name, local and unit ids are all set. See Figure S19. ○

4. OPERATING SYSTEM STRUCTURE DESIGN

The following three subsections describe the structure design processes of the FTDCS operating system. The operating system kernel processes are described in Section 4.1. Section 4.2 explains the executive processes. Finally, the distributed system manager structure design is given in Section 4.3.

4.1 Kernel Processes

The following processes are the operating system kernel functions. They are called by the simulator functions from Section 3. The corresponding structure diagrams are shown in Figures K1 through K15.

1.1 KB_boot

This process provides the boot entry point for the operating system. It initializes the OS through the predefined processor configuration data.

It invokes each kernel manager initialization entry point with configuration data ("KMB_cpu", "KMB_memory" and "KMB_link"). Next, buffer is allocated for the executive configuration data ("K_new_buffer"), the executive boot entry point is invoked ("XB_boot") and the executive configuration data buffer is deallocated ("K_release_buffer"). If the DSM configuration data is present, then a new buffer is allocated for it ("K_new_buffer"), DSM configuration data is copied to the buffer and the buffer is submitted to the executive ("K_cpu_k2x"). Finally, interrupted execution is continued ("K_cpu_exit"). See Figure K1.

2.1 K_cpu_enter

This process is called by interrupt service routines to make a context switch to the operating system context after an interrupt. It blocks the execution of any active user process. See Figure K2. O

2.2 K_cpu_fork

This process is invoked by the interrupt function to add an event to the tail of the kernel event pending queue. See Figure K2. O

1.1.1 KMB_cpu

This process is the initialization entry point for the kernel processor manager. It allocates memory ("K_allocate") for processor control structure and sets it to that from configuration data. It allocates memory for the kernel and executive pending event queues and initializes them. Also, the user, kernel and executive flags are cleared. See Figure K1.

1.1.2 KMB_memory

This process is the initialization entry point for the kernel memory manager. It initializes memory management from configuration data and also initializes buffer, packet and queue management ("K_set_stack"). Finally, the executive accessible entry points are set ("set exec. entry pts."). See Figures K1 and K3.

1.1.3 KMB_link

This process is the initialization entry point for the kernel link manager. It allocates ("K_allocate") and initializes the link control structures. Next, it invokes all the link handler and server initialization entries. See Figures K1 and K4.

1.1.4 K_new_buffer

This process allocates a data buffer. See Figure K1. ○

1.1.5 XB_boot

This is the executive boot entry point process. It has been explained in Section 4.2, as process 2. See Figures K1 and E2.

1.1.6 K_release_buffer

This process deallocates a data buffer by decrementing the number of links to the buffer. See Figure K1. ○

1.1.7 K_cpu_k2x

This process is invoked by kernel functions to add an event to the executive event pending queue. The event is added to the end of the queue. See Figure K1. ○

1.1.8 K_cpu_exit

This process is called by interrupt service routines upon their exit, to return to the normal operation. If the interrupted execution is not the kernel mode but the executive mode, processor interrupts are enabled ("K86_enable") and the kernel mode is entered ("K_cpu_kernel"). The kernel busy flag is set if the kernel or executive events are pending ("kernel busy"). "Unblock user" is called to enable processor interrupts and unblock the user if a user process is pending. If none of the above conditions are true, idle state is entered ("idle state"). Finally, processor interrupts are enabled. See Figures K1 and K5.

1.1.1.1 K_allocate

This process allocates a contiguous block of memory with at least the specified size. See Figure K1. ○

1.1.2.1 K_set_stack

This process preallocates a number of blocks of a fixed size, allowing allocation and deallocation of these blocks with minimal overhead. After preallocation, if the number of blocks is exhausted, a preset number of blocks is again allocated. See Figure K3. ○

1.1.2.2. set exec. entry pts.

This process sets up the executive accessible entry points (i.e., executive functions which access kernel memory management functions). See Figure K3. ○

1.1.3.1 KSI_mb_master

This process is the initialization entry point for the Multibus master shared memory server. It allocates ("K_allocate") and initializes the Multibus master shared memory link control structures with the input configuration data. See Figure K4.

1.1.3.2 KHI_mb_master

This process is the initialization entry point for the Multibus master shared memory interrupt handler. It calls "init. master communication" to establish the interrupt trap function. See Figure K4.

1.1.3.3 KSI_mb_slave

This process is the initialization entry point for the Multibus slave shared memory server. It allocates ("K_allocate") and initializes the Multibus shared memory link control structures with the input configuration data. See Figure K4.

1.1.3.4 KHI_mb_slave

This process is the initialization entry point for the Multibus slave shared memory interrupt handler. It calls "init. slave communication" to establish the interrupt trap function. See Figure K4.

1.1.3.5 KSI_i188

This process is the initialization entry point for the INTEL iSBC 188/48 server. It allocates ("K_allocate") and initializes the 188-based resource link control structures with the input configuration data. See Figure K4.

1.1.3.6 KHI_i188

This process is the initialization entry point for the INTEL iSBC 188/48 interrupt handler. It calls "perform board test" to reset the 188/48 board and establish the interrupt trap function. See Figure K4.

1.1.8.1 K86_enable

This process enables the function of interrupt recognition by the operating system. See Figure K5. ○

1.1.8.2 K_cpu_kernel

This process implements the operating system running in the kernel mode. The kernel pending events are added to the kernel event queue and processor interrupts are enabled ("K86_enable"). For each event in the kernel event queue, the link manager in entry point is invoked ("K_link_in"). Processor interrupts are disabled ("K86_disable"). The above steps are repeated until no more events are pending in the kernel event queue. Finally, the executive mode is entered ("K_cpu_executive"). See Figures K5 and K6.

1.1.8.3 kernel busy

If the kernel or executive events are pending, this process sets the kernel busy flag to true. See Figure K5. ○

1.1.8.4 unblock user

This process enables processor interrupts and unblocks the user process. See Figure K5. ○

1.1.8.5 enter idle

This process enables processor interrupts and enters the idle state. See Figure K5. ○

1.1.3.2.1 init. master communication

This process establishes the interrupt trap function and initializes master communication. See Figure K4. ○

1.1.3.4.1 init. slave communication

This process establishes the interrupt trap function and initializes slave communication. See Figure K4. ○

1.1.3.6.1 perform board reset

This process resets the 188/48 board and establishes the interrupt trap function. See Figure K4. O

1.1.8.2.1 K_link_in

This process is the kernel link manager in entry point. It looks up the kernel, local and server ids from the respective tables. With the ids, the appropriate server in entry point ("kmb_master_in", "kmb_slave_in" or "k188_in") is invoked. See Figures K6 and K7.

1.1.8.2.2 K86_disable

This process disables the function of interrupt recognition by the operating system. See Figure K6. O

1.1.8.2.3 K_cpu_executive

This process implements the operating system running in the executive mode. Initially, it enables processor interrupts ("K86_enable"). For each event in the executive event queue, the executive resource manager in entry point is invoked ("XM_in"). Next, interrupts are disabled ("K86_disable"). The above steps are repeated until there are no more events in the executive event queue. Finally, if an application process is pending, the process is unblocked. Otherwise, idle state is entered. See Figures K6 and K8.

1.1.8.2.1.1 kmb_master_in

This process is the in entry point for the Multibus master shared memory server. Three interrupts are processed by this process.

For the transmit complete interrupt (which indicates that the slave has completed transmission to the master), current network packet data is submitted to the executive ("K_cpu_k2x").

The second type of interrupt is the transmit ready interrupt. A network packet is allocated ("K_new_network") and the slave data header is copied to it. A new slave to master buffer is allocated ("K_new_buffer") and receive ready is set in reply signal word.

For the receive complete interrupt, the master to slave buffer is deallocated ("K_release_buffer"). If there is a packet in the transmit queue ("K_next_queue"), the packet header and data are copied to the master to slave packet and the network packet is freed ("K_free_network"). Also transmit ready is set in reply signal word.

Finally, if the reply signal was set, reply interrupt is sent to the slave ("send reply to slave"). See Figures K7 and K9.

1.1.8.2.1.2 kmb_slave_in

This process is the in entry point for the Multibus slave shared memory server. Two types of interrupts are processed by this process.

For the transmit ready interrupt (which indicates that master has a transmission for the slave), a network packet ("K_new_network") and network packet data buffer ("K_new_buffer") are allocated. The network packet data is submitted to the executive ("K_cpu_k2x").

If the interrupt is receive ready, the master to slave buffer is deallocated ("K_release_buffer"). Also, the network packet is deallocated ("K_free_network") and transmit done is set in reply signal word. If there is a packet in the transmit queue ("K_next_queue"), it is copied to the slave to master packet and transmit ready is set in reply signal word.

Finally, if the reply signal was set, reply interrupt is sent to the master. See Figures K7 and K10.

1.1.8.2.1.3 k188_in

This process is the in entry point for the iSBC 188/48 server. If the interrupt type is receive data, "k188_receive" is called. Otherwise, (for the transmit complete interrupt), "k188_transmit" is called. If the carrier is detected, the link status is enabled. The link status is disabled for a lost carrier. See Figures K7 and K11.

1.1.8.2.3.1 XM_in

This is the executive resource manager in entry point. It has been explained in Section 4.2, as process 1. See Figures K8 and E1.

1.1.8.2.1.1.1 K_new_network

This process allocates a network data packet. See Figure K9. ○

1.1.8.2.1.1.2 K_next_queue

This process returns an item from the head of the queue. See Figure K9. ○

1.1.8.2.1.1.3 K_free_network

This process deallocates a network data packet. See Figure K9. ○

1.1.8.2.1.1.4 send reply to slave

This process sends a reply interrupt to the slave. See Figure K9. ○

1.1.8.2.1.2.1 send reply to master

This process sends a reply interrupt to the master. See Figure K10. ○

1.1.8.2.1.3.1 k188_transmit

This process processes an interrupt from the 188/48 board indicating that the data transmission is complete. If there are any transmissions pending for the link, then the input characters are processed ("in_188_raw") and characters are echoed to 188/48 board ("k188_tx_packet"). If an I/O packet is in the output queue, the output characters are processed ("out_188_raw"), I/O packet data buffer is deallocated ("K_release_buffer") and the I/O packet is deallocated ("K_free_io"). See Figure K11.

1.1.8.2.1.3.2 k188_receive

This process processes an interrupt from the 188/48 board indicating that the input data is available. It processes the input characters ("in_188_raw") and if there are characters to echo, they are echoed to 188/48 board ("k188_tx_packet"). Finally, a receive complete control packet is created and sent to the 188/48 board. See Figure K11.

1.1.8.2.1.3.1.1 in_188_raw

This process processes input characters from a raw input queue to an input data packet. For each character in the raw input queue, the character is copied to the input packet data and the echo buffer, and the input packet is submitted to the executive ("K_cpu_k2x"). Also, a new input packet ("K_new_io") and data buffer for input packet ("K_new_buffer") are allocated. See Figures K11 and K12.

1.1.8.2.1.3.1.2 k188_tx_packet

This process creates transmit control packet for the 188/48 board, copies output data to the 188/48 board and sends a control packet initiating data transmission. See Figure K11. ○

1.1.8.2.1.3.1.3 out_188_raw

This process processes output characters to the 188/48 board. A buffer is allocated for processed characters ("K_new_buffer") and characters in output data are copied to output buffer. Processed output is sent to 188/48 board ("k188_tx_packet") and the output buffer is deallocated ("K_release_buffer"). See Figures K11 and K13.

1.1.8.2.1.3.2.1 K_free_io

This process deallocates an I/O data packet. See Figure K11. ○

1.1.8.2.1.3.1.1.1 K_new_io

This process allocates an I/O data packet. See Figure K12. ○

3. K_link_out

This process is the out entry point for the kernel link manager. It finds the local and the server ids using the kernel id, and invokes the appropriate server out entry point ("kmb_master_out", "kmb_slave_out" or "k188_out").

Note: this process is the same as process 1.3.1.2 in Section 4.2. See Figures K14 and E12.

4. K_link_assign

This process is the assign entry point for the kernel link manager. It accesses the link configuration data from the link id and the server control structure from the link configuration data. The appropriate server assign entry point is then invoked ("kmb_master_assign", "kmb_slave_assign" or "k188_assign").

Note: this process is the same as process 2.6.1.2 in Section 4.2. See Figures K15 and E22.

3.1 kmb_master_out

This process is the out entry point for the Multibus master shared memory server. If the network link is busy, a new network packet is allocated, the packet data is copied to it ("K_copy_buffer"), and the new packet is added to the link transmit packet queue ("K_add_queue"). Otherwise, packet data is copied to the master to slave data ("K_copy_buffer") and transmit ready interrupt is sent to the slave ("send transmit ready to slave"). See Figure K14.

3.2 kmb_slave_out

This process is the out entry point for the Multibus slave shared memory server. It allocates a new network packet ("K_new_network"), copies the packet data and header to the new packet ("K_copy_buffer"). If the network link is busy transmitting, the packet is queued ("K_add_queue"). Otherwise, the packet is transmitted immediately ("send transmit ready to master"). See Figure K14.

3.3 k188_out

This process is the out entry point for the iSBC 188/48 server. If the output link is not busy, output data is processed immediately ("out_188_raw"). If the output link is busy, a new I/O packet is allocated ("K_new_io"), output packet data is copied to it ("K_copy_buffer") and the new packet is queued ("K_add_queue"). See Figure K14.

4.1 kmb_master_assign

This process is the assign entry point for the Multibus master shared memory server. It allocates ("K_allocate") and initializes a Multibus link control structure. It also allocates ("K_new_network") master/slave packets. Finally, the kernel link manager control entry point is invoked ("K_link_control"). See Figure K15.

4.2 kmb_slave_assign

This process is the assign entry point for the Multibus slave shared memory server. It allocates ("K_allocate") and initializes a Multibus link control structure and invokes the kernel link manager control entry point ("K_link_control"). See Figure K15.

4.3 k188_assign

This process is the assign entry point for the iSBC 188/48 server. It allocates ("K_allocate") and initializes a link control structure. A new buffer is allocated ("K_new_buffer") for raw and processed input data. An I/O packet is allocated for input data ("K_new_io"). Finally, the kernel link manager control entry point is invoked ("K_link_control"). See Figure K15.

3.1.1 K_copy_buffer

This process copies a data buffer by incrementing its link count, which ensures that the buffer is not deallocated until all copies are deallocated. See Figure K14. ○

3.1.2 K_add_queue

This process adds an item to the tail of a queue. See Figure K14. ○

3.1.3 send transmit ready to slave

This process sends transmit ready interrupt to the slave. See Figure K14. ○

3.2.1 send transmit ready to master

This process sends transmit ready interrupt to the master. See Figure K14. ○

4.1.1 K_link_control

This process is the control entry point for the kernel link manager. If the kernel id is empty, "kl_enter" is called to assign a new kernel id. Otherwise, it finds the local and the server id using the kernel id, and invokes the appropriate server control entry point ("kmb_master_control", "kmb_slave_control" or "k188_control). See Figure K15.

4.1.1.1 kmb_master_control

This process is the control entry point for the Multibus master shared memory server. Currently no control functions are implemented. See Figure K15. ○

4.1.1.2 kmb_slave_control

This process is the control entry point for the Multibus slave shared memory server. Currently no control functions are implemented. See Figure K15. ○

4.1.1.3 k188_control

This process is the control entry point for the iSBC 188/48 server. Currently no control functions are implemented. See Figure K15. ○

4.1.1.4 kl_enter

This process assigns a new kernel id. Entries are made in the kernel id to server id, kernel id to local id and unit id to kernel id tables. See Figure K15. ○

5. K_free_user

This process deallocates an application consumer packet. See Figure E14. ○

6. K_free_buffer

This process deallocates a data buffer. See Figure E18. ○

4.2 Executive Processes

The following describes the processes identified in the operating system executive. The corresponding structure diagrams are shown in Figures E1 through E42.

1. XM_in

This process is the in entry point for the executive manager. If the executive id is empty, the executive control in entry point is invoked ("xcontrol_in"). Otherwise, the process finds the resource manager and the local id, and invokes the appropriate resource manager in entry point ("xnetwork_in", "xio_in" or "xprocess_in"). See Figure E1.

2. XB_boot

This process is the boot entry point for the executive. It sets up the executive memory management ("X_set_memory") and allocates memory for the executive data structures. Next, it sets up the executive message management ("X_set_messages") and executive queue management ("X_set_queues"). Finally the initialization entry point for each executive component is invoked ("XCB_control", "XCB_router" and "XCB_manager"). See Figure E2.

1.1 xnetwork_in

This process is the in entry point for the network resource manager. It allocates an executive message ("X_new_message") and copies message data from network packet to it. Next, it invokes the executive router out entry point ("XR_out") to route the message. Finally, the network data packet is deallocated ("X_free_network"). See Figures E1 and E3.

1.2 xio_in

This process is the in entry point for the I/O resource manager. It calls "X_next_queue" to find out if there is a query message in the consumer request queue. If there is one, a reply message is created, the executive router in entry point ("XR_in") is invoked and the I/O data packet is deallocated ("X_free_io").

If the resource consumer has a junction branch, an executive message is allocated ("X_new_message"), a send message is created, the message is routed according to junction branch 0 ("X_route_junction") and the I/O data packet is deallocated ("X_free_io").

If there is neither a query message in the consumer request queue nor a junction branch for the resource consumer, the I/O packet is added to the resource consumer input queue ("X_add_queue"). See Figures E1 and E4.

1.3 xprocess_in

This process is the in entry point for the application resource manager. Based on the type of the application packet, the appropriate process is called to process the packet (e.g., for an application packet of type ACCEPT, "process_accept" is called). An error message is displayed for an invalid packet type ("X_report_error"). See Figures E1 and E5.

1.4 xcontrol_in

This process is the entry point for the executive control component. It is currently used to pass the DSM configuration data at boot time. If the DSM has a local name, a message is allocated ("X_new_message") and its source, destination and data are set. Finally, the executive manager out entry point ("XM_out") is invoked. See Figures E1 and E6.

2.1 X_set_memory

This process initializes the executive memory management. If the executive memory pool is to be expanded, "X_more_memory" is called. In case of a memory management error, "X_mem_error" is called. See Figure E2.

2.2 X_set_messages

This process initializes executive message management by preallocating a number of executive messages. See Figure E2. ○

2.3 X_set_queues

This process initializes executive queue management. See Figure E2. ○

2.4 XCB_control

This process is the initialization entry point for the executive control component. First, it allocates memory for DSM query control ("X_set_stack"). Next, the DSM query queue is initialized and executive control entry points are set. For each executive consumer, the router assign entry point ("XR_assign") is invoked in order to assign the consumer. If the DSM consumer name is assigned to executive, the router control entry point ("XR_control") is invoked to enter the local name. See Figures E2 and E7.

2.5 XCB_router

This process is the initialization entry point for the executive router component. It allocates and initializes ("X_allocate" and "X_set_stack") the general executive routing structures such as the consumer index table, name index table, etc. The router entry points are set. Finally, for each routing algorithm supported (in this case - simple and

NMR), the algorithm initialization entry point is invoked ("XRI_simple" and "XRI_nmr"). See Figures E2 and E8.

2.6 XCB_manager

This process is the initialization entry point for the executive manager component. It allocates and initializes ("X_allocate") the general executive routing structures such as resource manager control structures, executive id to manager mappings, executive id to local id mappings, etc. It then invokes the specific manager initialization entry points ("XMI_network", "XMI_io" and "XMI_process") for each resource manager supported by the operating system. See Figures E2 and E9.

1.1.1 X_new_message

This process allocates an executive message. See Figure E3. ○

1.1.2 XR_out

This process is the out entry point to the executive router. If the message destination is undefined, an error message is reported ("X_report_error"). If the message destination is external, the message is relayed appropriately ("XM_out").

If neither of the above cases is true, the process tries to find the source name from the routing name table. If the source name is undefined, it is entered in the table ("new_name"), a message is added to the out pending queue ("X_add_queue"), a system unknown consumer command is allocated ("X_next_query") and the query is issued to the DSM ("XC_query_dsm"). If the source name status is out pending, a message is added to the out pending queue ("X_add_message"). Otherwise, the appropriate routing algorithm out entry point is invoked. See Figures E3 and E10.

1.1.3 X_free_network

This process deallocates a network data packet by calling the kernel deallocate network packet entry point ("K_free_network"). See Figure E3.

1.2.1 X_next_queue

This process returns an item from the head of a queue. See Figure E4. ○

1.2.2 XR_in

This process is the in entry point to the executive router. Each message has to have the appropriate routing algorithm applied to it before it can be sent to the external world. This process invokes the executive control out entry point if the message destination is empty and the appropriate routing algorithm in entry point otherwise. See Figures E4 and E11.

1.2.3 X_free_io

This process deallocates an I/O consumer packet by calling the kernel deallocate I/O packet entry point ("K_free_io"). See Figure E4.

1.2.4 X_route_junction

This process routes a message to a specified junction branch by routing a copy of the message to the consumer associated with each route of the branch. For each route in the junction branch, if it is not the last route, the input message is copied to the route message ("X_copy_message"). The router in entry point is invoked to route the message. See Figure E4.

1.2.5 X_add_queue

This process adds an item to the tail of the queue. See Figure E4. ○

1.3.1 process_accept

This function processes an ACCEPT packet from an application resource consumer. If the consumer has a waiting query message ("X_next_queue"), the packet message is set to the data from the query message, the packet is added to the consumer ready queue ("X_add_queue") and the query message is deallocated ("X_free_message"). Finally, the kernel link manager out entry point ("K_link_out") is invoked in order to send an empty acknowledgement packet to the calling process to prevent it from blocking. See Figures E5 and E12.

1.3.2 process_query

This function processes a QUERY packet from an application resource consumer. The consumer's reply message queue is checked ("match_query") and the kernel link manager out entry point ("K_link_out") is invoked in order to send an empty acknowledgement packet to the process to prevent it from blocking. See Figures E5 and E13.

1.3.3 process_reply

This function processes a REPLY packet from an application resource consumer. An executive message is allocated ("X_new_message"). The message type, source, signature, destination and data are set. The executive router in entry point ("XR_in") is invoked to route the reply. The application consumer packet is deallocated ("X_free_user"). The kernel link manager out entry point ("K_link_out") is invoked. See Figures E5 and E14.

1.3.4 process_call

This function processes a CALL packet from an application resource consumer. The consumer's reply message queue is checked ("match_query") and if a matching reply is available it used to satisfy the request. Otherwise, "process_ready" is called to check an application consumer's ready packet queue to see if a packet is available for the process. The application will block if no packets are ready. See Figures E5 and E15.

1.3.5 process_receive

This function processes a RECEIVE packet from an application resource consumer. If a send message is available for the application consumer ("X_next_queue"), the packet message is set to the data from the send message, the packet is added to the application ready queue ("X_add_queue") and the send message is deallocated ("X_free_message"). Finally, "process_ready" is called to check an application consumer's ready packet queue to see if a packet is available for the process. The application will block if no packets are ready. See Figures E5 and E16.

1.3.6 process_send

This function processes a SEND packet from an application resource consumer. If the packet specifies a consumer, then an executive message is allocated ("X_new_message"), the message type, source, signature, destination and data are set and the message is sent to that consumer ("XR_in").

If the packet does not specify a consumer and the junction branch as specified in the packet has at least one route, then an executive message is allocated ("X_new_message"), the message type, source, signature, destination and data are set and the message is routed to that junction branch ("X_route_junction"). In case both the above fail an error message is reported ("X_report_error").

Finally, the application consumer packet is deallocated ("X_free_user") and the kernel link manager out entry point ("K_link_out") is invoked. See Figures E5 and E17.

1.3.7 process_ready

This process checks an application consumer's ready packet queue to see if a packet is available for the process. If the function is invoked with a READY packet, the ready packet is deallocated ("X_free_user").

Next, if a packet is available in the application consumer ready queue ("X_next_queue"), it passed to the application consumer via the kernel out entry point ("K_link_out"). Also, if the packet has data, the packet data is deallocated ("X_free_buffer") and ready application consumer packet is deallocated ("X_free_user").

If a packet is unavailable, the application consumer scheduling is blocked ("process_run2wait"). See Figures E5 and E18.

1.3.8 X_report_error

This process creates an error report message and sends it to the DSM. It allocates message for an error message ("X_new_message"). The message type, source, signature, and destination are set. A buffer is allocated for the DSM error command ("X_new_buffer"). The error message is formatted and sent to the DSM ("XR_in"). See Figure E5.

1.4.1 XM_out

This process is the out entry point to the executive resource manager. If the executive id is empty, the executive control out entry point is invoked ("xcontrol_out"). Otherwise, the process finds the resource manager and the local id, and invokes the appropriate resource manager out entry point ("xnetwork_out", "xio_out" or "xprocess_out"). See Figures E6 and E19.

2.1.1 X_more_memory

This process expands the size of the executive memory pool by requesting memory from the kernel memory pool. See Figure E2. ○

2.1.2 X_mem_error

This process is invoked by the memory management functions when a memory management error occurs. Examples of such errors include: memory management structure corruption, attempts to reallocate or free unallocated memory, and no more memory faults. See Figure E2. ○

2.4.1 X_set_stack

This process preallocates a number of blocks of fixed size, allowing allocation and deallocation of these blocks with minimal overhead. See Figure E7. ○

2.4.2 XR_assign

This process is the assign entry point of the executive router. It is invoked to add a consumer to the routing consumer tables. First, a consumer is entered in consumer table ("new_consumer"). For each name associated with the consumer, if the name is not in the table, it is entered ("new_name") and if the name is not local, the executive manager assign entry point ("XM_assign") is invoked to assign a network path. Next, the appropriate routing algorithm assign entry point is invoked. Finally, the pending queues are checked for messages directed to the new names, and if any are found, these are routed as required ("XR_out"). See Figures E7 and E20.

2.4.3 XR_control

This process is the control entry point of the executive router. If the routing algorithm is empty, it calls one of the control processes below depending upon the control code. The control processes include "xr_enter_name" (to enter local executive id for name) and "xr_set_junction" (to reset consumer junction). Otherwise, the appropriate routing algorithm control entry point is invoked. Invalid codes however, cause an error message to be reported ("X_report_error"). See Figures E7 and E21.

2.5.1 X_allocate

This process allocates a contiguous block of memory having at least a specified size. See Figure E8. ○

2.5.2 XRI_simple

This process is the initialization entry point for the simple router. It sets simple routing entry points in the executive control structure. See Figure E8. ○

2.5.3 XRI_nmr

This process is the initialization entry point for the NMR router. It allocates memory for the NMR routing control ("X_allocate"). It also allocates memory for NMR consumers and messages ("X_set_stack"). Finally, it sets the NMR routing entry points in the executive control structure. See Figure E8.

2.6.1 XMI_network

This process is the initialization entry point for the executive network resource manager. Memory is allocated for the network resource control and link table ("X_allocate"). Next, for each network resource, its link is activated to enable the reception of network messages from that link ("XM_control" and "K_link_assign"). See Figures E9 and E22.

2.6.2 XMI_io

This process is the initialization entry point for the executive I/O resource manager. Memory is allocated for the I/O resource control, link table and resource consumer table ("X_allocate"). Next, for each I/O resource link, the link table entry is initialized. Also, the entry points in the executive manager control structure are set. See Figure E9.

2.6.3 XMI_process

This process is the initialization entry point for the executive application resource manager. Memory is allocated for the application resource control, link table and resource consumer table ("X_allocate"). Next, for each application resource link, the link table entry is initialized. Also, the entry points in the executive manager control structure are set. See Figure E9.

1.1.2.1 new_name

This process creates a new entry in the name routing tables and initializes it. A name structure is allocated. If the name id is greater than the name index, the latter is reallocated ("X_reallocate"). The name data structure is entered into the table and initialized. Finally, if the consumer id is defined, the name is linked to the consumer name list ("link name"). See Figures E10 and E23.

1.1.2.2 X_next_query

This process allocates a data buffer ("X_new_buffer") and initializes it as a system command by setting the command header parameters. See Figure E10.

1.1.2.3 XC_query_dsm

This process sends a query message to the DSM. It allocates a message for query ("X_new_message"). The message type, source, data, signature, and destination are set. Next, it is added to the outstanding query queue ("X_add_queue") and the message is sent by invoking the executive router in entry point ("XR_in"). See Figures E10 and E24.

1.1.2.4 X_add_message

This process adds a message to the out pending queue. See Figure E10. ○

1.1.2.5 invoke routing alg. out point

This process looks up the routing algorithm for the message, and invokes the out entry point of the appropriate routing algorithm. Currently, there are 2 routing algorithms being used: simple and NMR. Thus, this process calls either "xsimple_out" or "xnmr_out". See Figure E10.

1.2.2.1 xcontrol_out

This process is the out entry point for the executive control component. If the message is a reply message, it is sent to the DSM as a reply for the query ("query reply"). Otherwise, the message is processed locally as either a DSM or an executive command ("set command"). See Figures E11 and E25.

1.2.2.2 invoke routing alg. in point

This process looks up the routing algorithm for the message destination, and invokes the in entry point of the appropriate routing algorithm. Currently, there are 2 routing algorithms being used: simple and NMR. Thus, this process calls either "xsimple_in" or "xnmr_in". See Figure E11.

1.2.4.1 X_copy_message

This process allocates an executive message ("X_new_message") and copies the input message to it ("X_copy_buffer"). See Figure E4.

1.3.1.1 X_free_message

This process deallocates an executive message. See Figure E12. ○

1.3.1.2 K_link_out

This process is the same as the process 3 in Section 4.1 above. See Figures E12 and K14.

1.3.2.1 match_query

This process checks the application consumer reply message queue for a reply to a given query or call. If the queue contains a reply for the query ("X_find_signature"), packet message data is set to reply message, the packet is added to the application consumer ready packet queue ("X_add_queue") and the reply message is deallocated ("X_free_message").

Otherwise, the packet is added to the application consumer query packet queue ("X_add_queue") and a query message is sent as indicated by the packet message data ("send consumer query packet"). See Figures E13 and E26.

1.3.3.1 X_free_user

This process deallocates an application consumer packet by invoking the kernel deallocate application packet entry point ("K_free_user"). See Figure E14.

1.3.7.1 X_free_buffer

This process deallocates a data buffer by decrementing the number of links to the buffer. If the link count is 0, the kernel deallocate buffer entry point ("K_free_buffer") is called. See Figure E18.

1.3.7.2 process_run2wait

This process disables the scheduling of an application consumer, blocking that consumer. The next application consumer with scheduling enabled is scheduled ("next_process"). See Figure E18.

1.3.8.1 X_new_buffer

This process allocates a data buffer by invoking the kernel allocate buffer process ("K_new_buffer"). See Figure E5.

1.4.1.1 xnetwork_out

This process is the out entry point for the network resource manager. It creates a network packet from the message data and invokes the kernel out entry point ("K_link_out") to transmit the packet. If the message has data, the message data is deallocated ("X_free_buffer"). Finally, the executive message is deallocated ("X_free_message"). See Figures E19 and E27.

1.4.1.2 xio_out

This process is the out entry point for the I/O resource manager. If the message type is SEND, the send message is processed ("xio_send"), and if the message type is QUERY, the query message is processed ("xio_query"). For invalid message types, an error message is reported ("X_report_error"). See Figures E19 and E28.

1.4.1.3 xprocess_out

This process is the out entry point for the application resource manager. Depending upon the type of the message (SEND, QUERY or REPLY), the appropriate process is called to process the message ("process_out_send", "process_out_query" or "process_out_reply"). For invalid message types, an error message is reported ("X_report_error"). See Figures E19 and E29.

2.4.2.1 new_consumer

This process creates a new entry in the consumer routing tables, and initializes it. If the consumer id is greater than the consumer index size, space for the latter is reallocated ("X_reallocate"). See Figure E20.

2.4.2.2 XM_assign

This process is the assign entry point of the executive manager. It looks up the local resource id, and if one is found, the appropriate resource manager assign entry point is invoked ("xnetwork_assign", "xio_assign" or "xprocess_assign"). Otherwise, an error message is reported ("X_report_error"). See Figures E20 and E30.

2.4.2.3 invoke routing alg. assign entry pt.

This process invokes the assign entry point of the routing algorithm associated with that consumer. Currently, there are 2 routing algorithms being used: simple and NMR. Thus, this process calls either "xsimple_assign" or "xnmr_assign". See Figure E20.

2.4.3.1 xr_enter_name

This process assigns an executive identifier to a specified name. It looks up the name from the input name id. If the name is not defined, an error message is reported ("X_report_error"). Otherwise, the executive manager assign entry point ("XM_assign") is invoked. See Figure E21.

2.4.3.2 xr_set_junction

This process resets a consumer junction. With the executive id for the name, it invokes the executive manager control entry point. See Figure E21.

2.4.3.3 invoke routing alg. control entry pt.

This process invokes the appropriate control entry point of the routing algorithm. Currently, there are 2 routing algorithms being used: simple and NMR. Thus, this process calls either "xsimple_control" or "xnmr_control". See Figure E21.

2.6.1.1 XM_control

This process is the control entry point of the executive manager. If the executive id is not empty, the appropriate resource manager control entry point is invoked ("xnetwork_control", "xio_control" or "xprocess_control"). Otherwise, if the control code is ENTER, "xm_enter" is called to enter the executive id mappings. For all invalid codes an error message is reported ("X_report_error"). See Figures E22 and E31.

2.6.1.2 K_link_assign

This is the same as the process 4 in Section 4.1. See Figures E22 and K15.

1.1.2.1.1 X_reallocate

This process reallocates an allocated block of memory. See Figure E23. ○

1.1.2.1.2 enter and init. name

This process enters the name data structure in the name index by name id and initializes the name data structure fields. Also the name state is set. See Figure E23. ○

1.1.2.1.3 link name

This process links the new name to the consumer name list, if the consumer id is defined. See Figure E23. O

1.1.2.5.1 xsimple_out

This process is the out entry point to the simple router. The received messages are submitted to the executive resource manager for transmission to the appropriate active resource ("XM_out"). See Figure E10.

1.1.2.5.2 xnmr_out

This process is the out entry point to the NMR router. First, the NMR consumer message list is checked for another message copy. If no other copy is found, the first message copy is processed ("xnmr_out_new"). If the message copy has a valid message, it is processed ("xnmr_out_valid"). If the message copy does not have a valid message, a message copy stating that no valid message is available is processed ("xnmr_out_error"). Finally, if the message is the last expected copy, message reception is completed ("xnmr_done"). See Figures E10 and E32.

1.2.2.1.1 query reply

This process initially, finds a query for reply ("X_find_signature"). If the query is not found, an error message is reported ("X_report_error"). Otherwise, a notify function associated with the query is invoked, query request is freed, message data is deallocated ("X_free_buffer") and the message is deallocated ("X_free_message"). See Figure E25.

1.2.2.1.2 set command

This function processes the message data as a command. If the command is to DSM, it is copied ("X_copy_message") and relayed to the DSM ("XR_in"). Otherwise, the local executive command is processed ("X_command"). Finally, the message data is deallocated ("X_free_buffer") and the message is deallocated ("X_free_message"). See Figure E25.

1.2.2.2.1 xsimple_in

This process is the in entry point to the simple router. Each message is sent to its destination consumer ("X_route_consumer"). See Figure E11.

1.2.2.2.2 xnmr_in

This process is the in entry point to the NMR router. Each message is copied and sent to its destination consumer ("X_route_consumer"). See Figure E11.

1.2.4.1.1 X_copy_buffer

This process copies a data buffer by incrementing the link count to ensure that the buffer will not be actually deallocated until all copies are deallocated. See Figure E4. ○

1.3.2.1.1 send consumer query packet

This process sends a consumer query packet. If the packet specifies a consumer, an executive message is allocated ("X_new_message"), the type, signature, source, destination and data are set for the message and the reply is routed ("XR_in").

If the packet does not specify a consumer and the packet's junction branch has at least one route, then an executive message is allocated ("X_new_message"), the type, signature, source, destination and data are set for the message and the message is routed to the junction branch ("X_route_junction"). If neither of the above conditions are true, an error message is reported ("X_report_error"). See Figure E26.

1.3.3.1.1 K_free_user

This is the same as the process 5 in Section 4.1. See Figure E14. ○

1.3.7.1.1 K_free_buffer

This is the same as the process 6 in Section 4.1. See Figure E18. ○

1.3.7.2.1 next_process

This process checks if an application consumer is in the ready queue ("X_next_queue"). If so, the application consumer is scheduled ("K_link_out"). Also, if the scheduled application consumer has a ready packet with data, the packet data is deallocated ("X_free_buffer") and the application consumer packet is deallocated ("X_free_user"). See Figures E18 and E33.

1.4.1.2.1 xio_send

This function processes a SEND output message. If the message has associated data, the I/O packet data is set to it, kernel out entry point is invoked ("K_link_out"), and the message data is deallocated ("X_free_buffer"). Finally, the executive message is deallocated ("X_free_message"). See Figure E28.

1.4.1.2.2 xio_query

This function processes a QUERY output message for an I/O resource consumer. First, it checks if an I/O packet is in the data queue ("X_next_queue"). If so, a reply message is created, the message is routed ("XR_in") and I/O data packet is deallocated

("X_free_io"). If there is no I/O packet in the data queue, the query message is added to the resource consumer request queue ("X_add_queue"). See Figure E28.

1.4.1.3.1 process_out_query

This function processes a QUERY output message by checking if the application consumer has an outstanding accept packet. If the application consumer has an accept packet, the accept packet is added to the ready packet queue ("X_add_queue"), the executive query message is deallocated ("X_free_message") and the application consumer scheduling is enabled (if it was disabled) ("process_wait2ready"). If there is no accept packet, the query message is added to the query message queue ("X_add_queue"). See Figures E29 and E34.

1.4.1.3.2 process_out_reply

This function processes a REPLY output message by checking if the application consumer has a query or call packet outstanding for the reply. First, it checks if the application consumer has a query for reply ("X_find_signature"). If it does, the query packet is added to the ready packet queue ("X_add_queue"), the executive reply message is deallocated ("X_free_message") and the application consumer scheduling is enabled (if it was disabled) ("process_wait2ready"). If there is no query packet, the reply message is added to the reply message queue ("X_add_queue"). See Figures E29 and E35.

1.4.1.3.3 process_out_send

This function processes a SEND output message by checking if the application consumer has an outstanding receive packet. If the application consumer has a receive packet, it is added to the ready packet queue ("X_add_queue"), the executive send message is deallocated ("X_free_message") and the application consumer scheduling is enabled (if it was disabled) ("process_wait2ready"). If there is no receive packet, the send message is added to the send message queue ("X_add_queue"). See Figures E29 and E36.

2.4.2.2.1 xnetwork_assign

This process is the assign entry point of the network resource manager. If the link to resource is found and the network path unassigned, the executive manager control entry point ("XM_control") is invoked to assign executive id. Otherwise, an error message is reported stating that the resource is unavailable ("X_report_error"). See Figure E30.

2.4.2.2.2 xio_assign

This process is the assign entry point of the I/O resource manager. If the link to the resource is found, memory is allocated for the resource consumer structure ("X_allocate"), resource consumer table is expanded if needed ("X_reallocate"), executive manager control entry point is invoked to enter the consumer ("XM_control") and the kernel link assign entry point is invoked to activate the resource ("K_link_assign"). Otherwise, an error message is reported stating that the resource is unavailable ("X_report_error"). See Figures E30 and E37.

2.4.2.2.3 xprocess_assign

This process is the assign entry point of the application resource manager. If the link to the resource is found, memory is allocated for the resource consumer structure ("X_allocate"); resource consumer table is expanded if needed ("X_reallocate"); executive manager control entry point is invoked to enter the consumer ("XM_control"); kernel link assign entry point is invoked to activate the resource ("K_link_assign") and the resource consumer scheduling is enabled ("process_wait2ready"). Otherwise, an error message is reported stating that the resource is unavailable ("X_report_error"). See Figures E30 and E38.

2.4.2.3.1 xsimple_assign

This process is the assign entry point of the simple router. It allocates memory for the simple consumer local data ("X_allocate") and initializes the data which is specific to the simple routing algorithm. See Figure E20.

2.4.2.3.2 xnmr_assign

This process is the assign entry point of the NMR router. It initializes the data which is specific to the NMR routing algorithm. See Figure E20. ○

2.4.3.3.1 xsimple_control

This process is the control entry point of the simple router. An error message is reported for an invalid control code ("X_report_error"). Note: currently, no simple control functions are implemented. See Figure E21.

2.4.3.3.2 xnmr_control

This process is the control entry point of the NMR router. An error message is reported for an invalid control code ("X_report_error"). Note: currently, no NMR control functions are implemented. See Figure E21.

2.6.1.1.1 xm_enter

This process allocates a new executive id and makes entries for the resource manager and the local id in the manager mapping tables. If there are no more available executive ids, the tables are expanded ("X_reallocate"). See Figure E31.

2.6.1.1.2 xnetwork_control

This process is the control entry point of the network resource manager. An error message is reported for an invalid control code ("X_report_error"). Note: currently, no control functions are implemented. See Figure E31.

2.6.1.1.3 xio_control

This process is the control entry point of the I/O resource manager. If the control code is JUNCTION, "xi_reset_junction" is called to reset the I/O resource consumer junction. An error message is reported for an invalid control code ("X_report_error"). See Figure E31.

2.6.1.1.4 xprocess_control

This process is the control entry point of the application resource manager. If the control code is JUNCTION, "xp_reset_junction" is called to reset the application consumer junction. An error message is reported for an invalid control code ("X_report_error"). See Figure E31.

1.1.2.4.2.1 xnmr_out_new

This function processes the first copy of a message received from an NMR consumer. It initializes an NMR structure. If N is 1, the message is added to the valid copy queue ("X_add_queue") and a valid message is created ("xnmr_ready"). Otherwise the message is added to the error copy queue ("X_add_queue"). See Figure E32.

1.1.2.4.2.2 xnmr_out_error

This function processes a message from an NMR consumer when no valid copy of the message is available. First, it checks the error copy queue for the exact copy message. If the exact copy is found, it is added to the valid copy queue ("X_add_queue"); deleted from the error copy queue ("X_next_queue"). Next, if a valid copy is available, a valid message is created ("xnmr_ready"). Otherwise, the message is added to the error copy queue ("X_add_queue"). See Figure E32.

1.1.2.4.2.3 xnmr_out_valid

This function processes a message from an NMR consumer when a valid copy of the message is available. If the message same as the valid copy, it is added to the valid copy queue; otherwise it is added to the error copy queue ("X_add_queue"). See Figure E32.

1.1.2.4.2.4 xnmr_done

This process is invoked when the message reception is complete. If the NMR message has error copies, then for each copy in the error queue ("X_next_queue"), an error message is reported to the DSM ("X_report_error") and the error copy is added to the valid queue ("X_add_queue").

If valid copy count is not equal to expected copy count, then for each NMR consumer name, if a copy is found, the message and data are deallocated ("X_free_buffer" and "X_free_message"). Otherwise, an error message is reported to the DSM ("X_report_error") stating that the message is missing.

If valid copy count is equal to expected copy count, then for each message in the valid queue, the message and data are deallocated ("X_free_buffer" and "X_free_message"). See Figures E32 and E42.

1.2.2.1.1.1 X_find_signature

This process searches a queue for a message containing the given signature. See Figure E25. O

1.2.2.1.2.1 X_command

This function processes a system command directed to the executive. One of the following processes is called based on the control code: "X_command" (to process a list of commands); "XR_assign" (to add a consumer); "XR_control" (add a name); "list_status" (to report executive status) or "X_report_error" (to report an error for an invalid code). See Figure E25.

1.2.2.2.1.1 X_route_consumer

This process sends a message to a specified consumer ("XR_out") by sending a copy of the message ("X_copy_message") to each name associated with the consumer. See Figures E11 and E39.

1.4.1.3.1.1 process_wait2ready

This process enables the scheduling of an application consumer and adds it to the ready queue ("X_add_queue"). The next application process is scheduled ("next_process"). See Figure E34.

2.6.1.1.3.1 xi_reset_junction

This process resets the junction for an I/O resource consumer. It accesses the I/O resource consumer from the resource consumer table; reallocates the I/O resource consumer to hold new junction ("X_reallocate") and copies resource consumer junction from input junction. See Figure E31.

2.6.1.1.4.1 xp_reset_junction

This process resets the junction for an application resource consumer. It accesses the application resource consumer from the resource consumer table; reallocates the application consumer to hold new junction ("X_reallocate") and copies consumer junction from input junction. See Figure E31.

1.1.2.4.2.1.1 xnmr_ready

This process is called when a valid message is available from an NMR consumer. The valid message is copied ("X_copy_message") and is routed ("XM_out"). Next, for each NMR message in the NMR consumer message list, if the message destination is the same as the valid message and the message signature is less than the valid message signature by a predetermined constant, the message reception is completed ("xnmr_done"). See Figures E32 and E41.

1.2.2.1.2.1.1 list_status

This process creates and sends a reply message to a request for the executive status. A message data buffer ("X_new_buffer") and a message ("X_new_message") are allocated for the reply. The type, signature, source, destination and data are set for the reply message. Executive router in entry point ("XR_in") is invoked to route the reply message. See Figures E25 and E40.

4.3 Distributed System Manager & Shell Processes

The following describes the processes identified in the distributed system manager (DSM) of the operating system. The corresponding structure diagrams are shown in Figures D1 through D15.

Main (DSM)

This is the distributed system manager (DSM) mainline process. It calls "sys_accept" to set up the system to accept commands. Next, it repeats the two steps "sys_read" (i.e., to read commands from standard input) and "mgr_command" (to execute the command). See Figure D1.

1. sys_accept

This process calls "set user packet buffer" to set the user packet data and then calls "tx_packet" to send the user packet to the system. See Figure D1.

2. sys_read

This process reads command from standard input. See Figure D1. ○

3. mgr_command

Depending upon the type of the command, this process calls the respective process to execute the command. For example, "mc_boot" is called if the command is to initialize DSM with configuration boot data and "mc_define" is called to define a consumer and so on. See Figures D1 and D2.

1.1 set user packet data

This process sets the various fields in the data packet with the user packet data. The packet type is set to "UPKT_ACCEPT" and the data length to 0. Also set are the address of the reply buffer and its length. See Figure D1. ○

1.2 tx_packet

This process checks if the data packet has any data and if so, it calls "U_copy" to copy packet data to user packet buffer. Next, it calls "write to pipe to os" to write the user packet buffer to pipe to operating system. See Figure D1.

3.1 mc_list

For each command in the command list, this process calls the respective process to execute that command. If the command in the list is itself a list of commands, it calls itself. This process is repeated until all commands are executed. See Figures D2 and D3.

3.2 mc_boot

This process is the boot entry point for the distributed system manager. It performs the following functions. Memory management, stack for messages, buffer to write to execs and acknowledgement buffer are set up by a call to "set up memory, stack & buffer sizes". Next, "allocate spaces for tables" is called to allocate spaces for managers, routers, resources, executives and consumers tables. The system id is initialized to that in the configuration.

Each manager is then added to the manager table (repeated calls to "mc_add_manager"). Similarly, each router and resource is added to the corresponding table (repeated calls to "mc_add_router" and "mc_add_resource"). Not only is each executive added to the executive table, each executive's link is also added to the executive's link table (a call to "add_execs"). The DSM is added to the consumer table, and initialized with configuration data and each name required for DSM is entered in system and consumer tables ("enter_name & enter_consumer"). Executives with DSM resource are assigned to DSM consumer ("assign_resource"). Finally, space is allocated for junctions and the DSM consumer state is initialized. See Figures D2 and D4.

3.3 mc_unknown_con

This process adds a consumer address to an executive which must communicate with it. It gets the consumer table entry by calling "get consumer". Next, "assign_network" is called to assign networks between the consumer and the given executive. The process "prepare acknowledgement" is called to prepare an ack with "add consumer" command. Finally, the consumer data is added to the buffer ("add consumer"). See Figures D2 and D5.

3.4 mc_unknown_name

This process has not been implemented. See Figure D2.

3.5 mc_define

This process is used to define a consumer. It calls "find resource" to look up the resource from the system resource table. If the resource is not found, an error message is displayed. Else, a new consumer is created and entered into the system ("create & enter consumer"). See Figures D2 and D6.

3.6 mc_link

This process is used to define a link from one consumer to another. It calls "find consumer" to get the consumer (to be linked to this consumer) from the system consumer table. If the consumer is not found, an error message is displayed. Otherwise, process "link consumer" links the consumer. See Figures D2 and D7.

3.7 mc_run

This process is used to begin the execution of a consumer. It calls "find consumer" to get the consumer from the system consumer table. If the consumer is not found, an error message is displayed. Otherwise, the consumer's address is added to all executives which must communicate with it ("run_tos") and execution of all consumer names is begun ("run_names"). See Figures D2 and D8.

3.8 mc_get_consumer

This process sends an acknowledgement to the calling process with the status of the given consumer. It calls "find consumer" to get the consumer from the system consumer table. If the consumer is not found, it calls "prepare ack" to prepare an empty acknowledgement buffer. Otherwise, if the consumer does not have a junction, an empty one is allocated for it. Consumer data is placed in the buffer and an appropriate acknowledgement buffer is prepared. See Figures D2 and D9.

3.9 mc_get_cpu

This process sends an acknowledgement to the calling process with the status of the given executive. It calls "find executive" to get the id of the executive from the system executive table. If the executive is not found, it calls "prepare ack" to prepare an empty acknowledgement buffer. Otherwise, a request status command buffer is prepared and sent to the executive. See Figures D2 and D10.

3.10 mc_exec_error

This process prints out an error message with the given executive id and the error code. See Figure D2. O

3.11 mc_undefined

This process prints out a message indicating that the received command is undefined. See Figure D2. O

1.2.1 U_copy

This process copies a specified number of bytes from a source to a destination. See Figure D1. O

1.2.2 write to pipe to os

This process writes the user packet buffer to pipe to operating system. See Figure D1. ○

3.2.1 set up memory, stack & buffer sizes

This process sets up the following parameters required for system boot: memory management with 4 Kbytes, a stack for messages, a 256 byte buffer to write to execs, a 256 byte acknowledge buffer. See Figure D4. ○

3.2.2 allocate spaces for tables

This process allocates space for tables of the following: managers, routers, resources, executives, consumers and their names. See Figure D4. ○

3.2.3 mc_add_manager

This process allocates space for the new manager entry and initializes entry with manager configuration data. It also sets the resource count to 0 and adds the entry to the table of managers. See Figure D4. ○

3.2.4 mc_add_router

This process allocates space for the new router entry and initializes entry with router configuration data. The entry is added to the table of routers and the new entry index to the beginning of the system router list. See Figure D4. ○

3.2.5 mc_add_resource

This process allocates space for the new resource entry and its links, and initializes entry with resource configuration data. The entry is added to the table of resources and the new entry index to the beginning of the system resource list and resource manager's resource list. See Figure D4. ○

3.2.6 add execs

For each executive in the configuration, this process adds the executive to the executive table and the list ("mc_add_exec"), and adds all the executive's links to the executive's link table ("mc_add_link"). See Figure D4.

3.2.7 enter_name & enter_consumer

For each name required for the DSM, this process enters the name in system and consumer tables. Also, the DSM is added to the consumer table, and initialized with configuration data. See Figure D4. ○

3.2.8 assign_resource

This process assigns executives with the available resource to a consumer. The appropriate assign process ("assign_io" or "assign_appl") is called depending on the manager id. For any other id an error message is displayed. See Figures D4 and D11.

3.3.1 get consumer

This process gets the consumer name table entry from the given name id. It then gets the consumer table entry from the id of the consumer associated with the name. See Figure D5. ○

3.3.2 assign_network

This process assigns networks between the given executive assigned to the consumer and the consumer's other assigned executives.

The process gets the executive table entry for the given executive id. If the executive is in the consumer's name mask, it is removed from the mask. Next, "check & assign network" is called to test network executive and consumer name masks and if possible, assign a network. If no network is assigned at the end of all the testing and assigning, an error message is displayed. Finally, the network manager's resource list is sorted ("sort_networks"). See Figures D5 and D12.

3.3.3 prepare acknowledgement

This process prepares an appropriate acknowledgement buffer. See Figure D5. ○

3.3.4 add consumer

This process places the consumer id, router type, the name count and the consumer name list in the buffer. See Figure D5. ○

3.5.1 find resource

This process looks up the resource from the system resource table to get the id. See Figure D6. ○

3.5.2 display error msg

This process displays the error message given to it as its input (e.g., resource not found message, etc.). See Figure D6. ○

3.5.3 create & enter consumer

This process creates a new consumer entry. It calls "find router" to get the router id. If the router is not found, an error message is displayed. Otherwise "enter new consumer" is called to do further processing. See Figure D6.

3.6.1 find consumer

This process looks up consumer in the system resource table to get the id. See Figure D7. ○

3.6.2 link consumer

This process initially gets the consumer table entry and the junction data. It allocates space for junction and initializes it. For each route up to the given junction route count, it calls "get junction to destination" which tries to get a link from the source to the destination. Next, for each route in junction, a junction is set up between the consumer and the destination ("set_junction"). Finally, if the consumer already had junctions defined, the old junction space is freed and the new one is reset if the consumer is active ("free old & reset new junction"). See Figure D7.

3.7.1 run_tos

This process adds the consumer address to all executives which must communicate with the given consumer. First, it gets all executives which will communicate with the consumer, excluding those on which the consumer's names will execute. Next, the consumer id, router type and name count data are filled into a buffer.

For each executive and for each name in the consumer's name list, the name's id is placed in the command buffer. If the name is on the current executive, "name on current exec." is called. Otherwise, "add consumer to exec." will add the consumer address to the executive. Finally a command is issued to the current executive. See Figures D8 and D13.

3.7.2 run_names

This process starts the execution of each name for a given consumer, after "run_tos" (above) has established all links. See Figure D8. ○

3.8.1 place consumer data in buffer

This process places the consumer data such as: id, router type, name count, link data and junction data in the buffer. See Figure D9. ○

3.9.1 find executive

This process looks up the executive name in the system executive table to get its id. See Figure D10. ○

3.9.2 prepare & send reply to executive

This process gets the system message from the stack. It then prepares a request status command buffer to send to the executive. Finally, it sends a query with cpu report as the reply function to the executive. See Figure D10. ○

3.2.6.1 mc_add_exec

This process adds a new executive entry to the executive list. It allocates space for the new entry and initializes it with executive configuration data. Next it adds the entry to the executives table and adds the entry index to the beginning of the system executive list. Finally, it calls "enter_name & enter_consumer" to add a consumer entry for the executive and a name entry for the executive consumer into the respective tables. See Figure D4.

3.2.6.2 mc_add_link

This process gets the id of the manager for the resource to be linked. It either calls the appropriate link process ("network_link", "io_link" or "appl_link") for a valid manager id or displays an error message for an invalid id. See Figures D4 and D14.

3.5.3.1 find router

This process looks up the given router name in the system router table to get the id. See Figure D6. ○

3.5.3.2 enter new consumer

Each consumer name is entered into the system and consumer tables (repeated calls to "enter_name & enter consumer"). Next, executives with available resource are assigned to the consumer ("assign_resource"). If all names were not successfully assigned, an error message indicating a resource limitation is displayed. Networks are established between executives of the consumer ("assign_network"). Finally, the new consumer is entered into the table. See Figure D6.

3.6.2.1 get junction to destination

For each route up to the given junction route count, this process looks up the destination consumer in the table ("find consumer"). If the consumer was not found, an error message is displayed. Otherwise, "set up junction list from branch" gets junction list to destination. See Figure D7.

3.6.2.2 set_junction

This process assigns networks between the executives of the given and the destination consumers. Note that the cases where the destination and the source reside on the same executive are excluded. See Figure D7. ○

3.6.2.3 free old & reset new junction

This process frees the space allocated for the old junction definition. If the consumer is active, the junction is reset. See Figure D7. ○

3.2.8.1 assign_io

This process assigns executives with available I/O resource to a consumer. First it calls "get resource" to get required I/O resource entry from the corresponding table. For each consumer's name, "find exec. for resource" tries to find the executive linked to the given resource. If an executive is found, it is assigned to the consumer ("assign exec. to consumer") and the executive table is sorted ("sort_execs"). See Figures D11 and D15.

3.2.8.2 assign_appl

This process assigns executives with available process resource to a consumer. As can be seen in the code for this process (FTDCS Operating Systems Revisions: Source Code Listings), it is exactly the same as that of "assign_io" making the exact same calls. See Figure D11. ○

3.3.2.1 check & assign network

For each executive in the network's executive and consumer's name masks, and for each name in the consumer's name list, this process places the network id in executive's name map, if name's executive is in both network's executive and consumer's name masks. The network load count is incremented. See Figure D12. ○

3.3.2.2 sort_networks

This process does a partial sort of networks in the system resource table, based on their load counts. Networks are sorted in the increasing order of their load counts, so that when new consumers are defined, the one with the smallest load is used. See Figure D12. ○

3.7.1.1 name on current exec.

Since the consumer name is on the current executive, its address is not added to the executive. The local and unit ids in the command buffer are both placed as EMPTY. See Figure D13. ○

3.7.1.2 add consumer to exec.

This process places the local id for current executive in the command buffer. If the local id is not empty, the link for name's executive and resource is also placed in the buffer. See Figure D13. ○

3.2.6.2.1 network_link

This process adds network link entry to an executive's link list. It gets the network resource entry from system resource table and the executive entry from the system executive table. It then increments the network's executive count if network resource mask does not include this executive. It adds the executive to the network's executive mask and the executive id to the network's link list. Finally, it calls "add all other execs" to connect all executives connected to this executives network. See Figure D14.

3.2.6.2.2 io_link

This process adds I/O link entry to an executive's link list. It gets the I/O resource entry from system resource table and the executive entry from the system executive table. It then increments the resource's executive count if IO resource mask does not include this executive. Finally, it adds the executive to the resource's executive mask and the executive id to the resource's link list. See Figure D14. ○

3.2.6.2.3 appl_link

This process adds a process link entry to an executive's link list. It gets the process resource entry from system resource table and the executive entry from the system executive table. It then increments the resource's executive count if the process resource mask does not include this executive. Finally, it adds the executive to the resource's executive mask and the executive id to the resource's link list. See Figure D14. ○

3.2.8.1.1 get resource

This process gets the resource table entry for the consumer's required resource. See Figure D15. ○

3.2.8.1.2 find exec. for resource

This process finds an executive linked to the required resource and in given mask. See Figure D15. ○

3.2.8.1.3 assign exec. to consumer

This process inserts data for the executive into the name list entry. The name is added to the beginning of the executive's name list and the executive to consumer's mask. Finally, the assign and load counts are incremented. See Figure D15. ○

3.2.8.1.4 sort_execs

This process does a partial sort of executives in the system executive table, based on their load counts. Executives are sorted in the increasing order of their load counts, so that when new consumers are defined, the one with the smallest load is used. See Figure D15. ○

3.6.2.1.1 set up junction list from branch

If the route's branch id is different from the previous, a junction list from the branch is set up. The branch's route base is initialized to the route count and the latter set to 0. See Figure D7. ○

3.2.6.2.1.1 add all other execs

For each entry in the system executive list, this process gets the executive entry from the system executive table. If this executive is assigned to the network, then all other executives in network's executive mask are added to this executive's executive mask. See Figure D14. ○

In addition to the above processes, there is a shell application process connected to a system console. This provides an interface to the DSM, in order to issue commands to the operating system and to display the system status reports. The corresponding structure diagrams are shown in figures H1 through H8.

Main (shell)

This is the mainline for the operating system shell. It prompts the user for a shell command, and when the user has entered a line, the appropriate routine is called to process it. A list of commands is displayed ("display commands"), if the user inputs a wrong command. See Figure H1.

1. sh_define

This function processes a "define consumer" shell command. It reads the consumer, resource, router names and name count, places the define consumer command in the buffer and sends the buffer to the DSM ("sys_command"). See Figure H1.

2. sh_link

This function processes a "link consumer" shell command. It reads the consumer name for each branch and route, places the link consumer command in the buffer and sends the buffer to the DSM ("sys_command"). See Figure H1.

3. sh_run

This function processes a "run consumer" shell command. It reads the consumer name, places the run consumer command in the buffer and sends the buffer to the DSM ("sys_command"). See Figure H1.

4. sh_status

This function processes a "status" shell command. It reads the choice of the status information and calls the appropriate process. A list of choices is displayed ("display choices") if the user specifies a wrong choice. See Figures H1 and H2.

5. display commands

This process displays a list of shell commands available and how to invoke each of them. See Figure H1. ○

1.1 sys_command

This process sends a command to the DSM. If an acknowledgement buffer is not provided, the command is sent to the DSM ("sys_write"). Otherwise, the command is sent to the DSM and the acknowledgement read back ("sys_call"). See Figures H1 and H3.

4.1 exec_status

This function processes a "status cpu" shell command. First, it reads the name of the CPU and prepares the command to get the executive status information. Next, it sends the command to the DSM and waits for a response ("sys_call"). If the reply buffer is not empty, the executive status information is displayed ("display exec. status"). Otherwise, a message is displayed indicating the failure to find the executive ("display exec. not found"). See Figures H2 and H4.

4.2 name_status

This process shows the name status information. It reads the consumer name, and prepares the command to get the name status information. Next, it sends the command to the DSM and waits for a response ("sys_call"). If the reply buffer is not empty, the name status information is displayed ("display name status"). Otherwise, a message is displayed indicating the failure to find the name ("display name not found"). See Figures H2 and H5.

4.3 map_status

This process shows the executive network map information. It reads the executive, and prepares the command to get the executive network map status information. Next, it sends the command to the DSM and waits for a response ("sys_call"). If the reply buffer is not empty, the executive network map status information is displayed ("display map status"). Otherwise, a message is displayed indicating the failure to find the executive ("display map not found"). See Figures H2 and H6.

4.4 consumer_status

This function processes a "status consumer" shell command. First, it reads the name of the consumer and prepares the command to get the consumer status information. Next, it sends the command to the DSM and waits for a response ("sys_call"). If the reply buffer is not empty, the consumer status information is displayed ("display consumer status"). Otherwise, a message is displayed indicating the failure to find the consumer ("display consumer not found"). See Figures H2 and H7.

4.5 display choices

This process displays a list of choices whose status information can be shown, and how to invoke them. See Figure H2. ○

1.1.1 sys_write

This process sends a message to another consumer. See Figure H3. ○

1.1.2 sys_call

This process sends a query message to a consumer, and waits for reply. It sends the user packet to the specified channel ("tx_packet"), puts the process in a ready state and waits for a reply ("wait_event"). See Figures H3 and H8.

4.1.1 display exec. status

This process displays the executive status information. This includes: executive memory, router and manager information; also the kernel memory information. See Figure H4. ○

4.1.2 display exec. not found

This process displays a message which states that the CPU (input by the user) was not found. See Figure H4. ○

4.2.1 display name status

This process displays the name status information. This includes: consumer, executive and resource names, and its external addresses. See Figure H5. ○

4.2.2 display name not found

This process displays a message which states that the name (input by the user) was not found. See Figure H5. ○

4.3.1 display map status

This process displays the executive network map information. See Figure H6. ○

4.3.2 display map not found

This process displays a message which states that the executive name (input by the user) was not found. See Figure H6. ○

4.4.1 display consumer status

This process displays the consumer status information. This includes: the consumer's router type, its executive, its name on the executive, the number of branches and routes it has, etc. See Figure H7. ○

4.4.2 display consumer not found

This process displays a message which states that the consumer name (input by the user) was not found. See Figure H7. ○

1.1.2.1 wait_event

This process is called whenever the calling process is ready to accept input messages. It waits for a receive packet from the operating system ("rx_packet"). If a notification function is indicated, it is invoked. Otherwise, the ready state is cleared. Finally, even after packet is processed, if the process is still in ready state, a user packet is sent to the operating system ("tx_packet"). See Figure H8.

1.1.2.1.1 rx_packet

This process is used to receive a user packet from the operating system to a process. The user packet is read from pipe from operating system. If the packet has reply data, it is copied from the user packet buffer. See Figure H8. ○

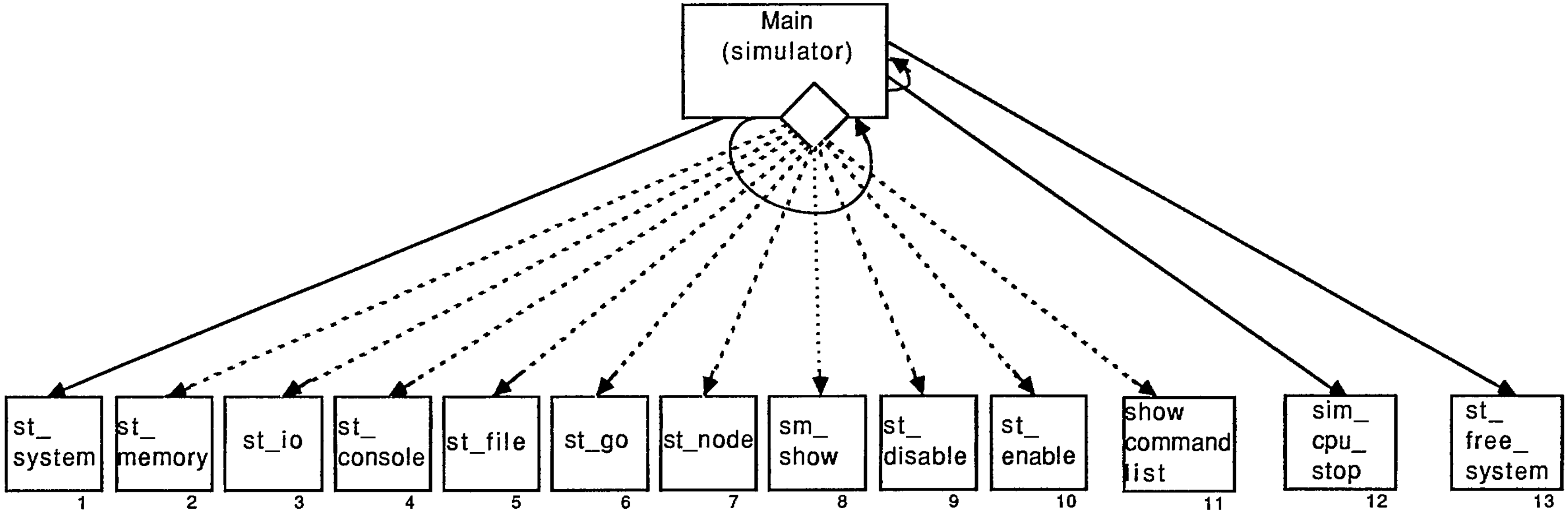


Figure S1: Main (Simulator)

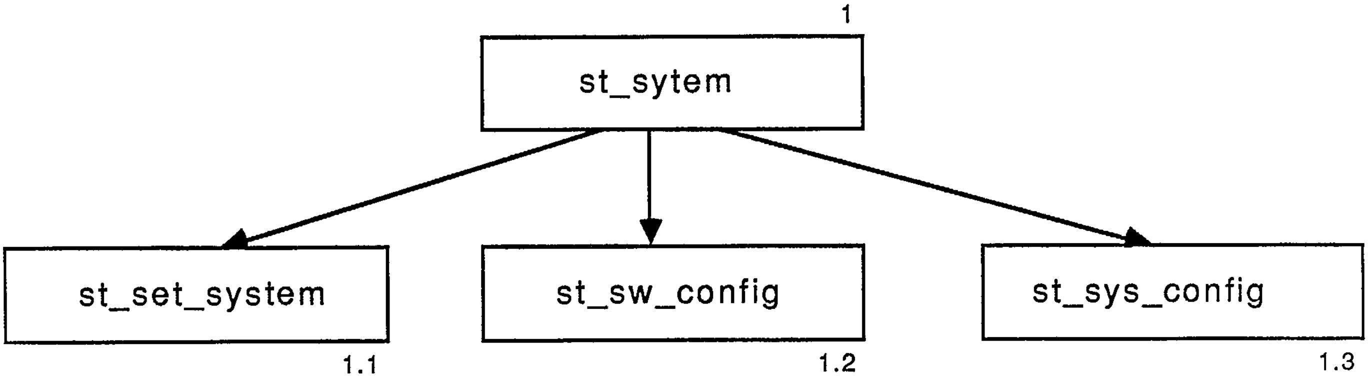


Figure S2: st_system

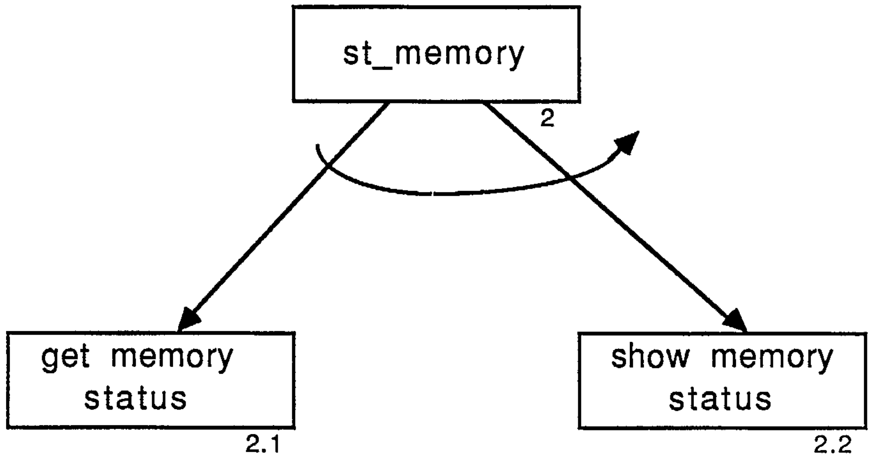


Figure S3: st_memory

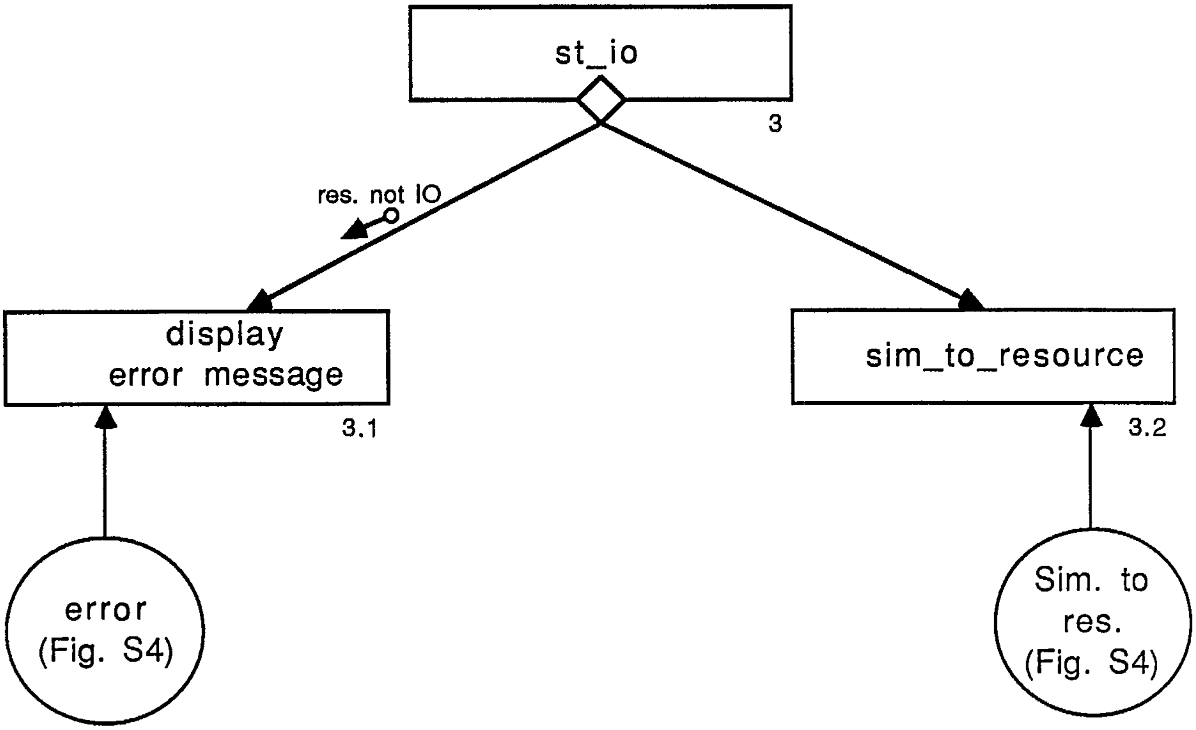


Figure S4: st_io

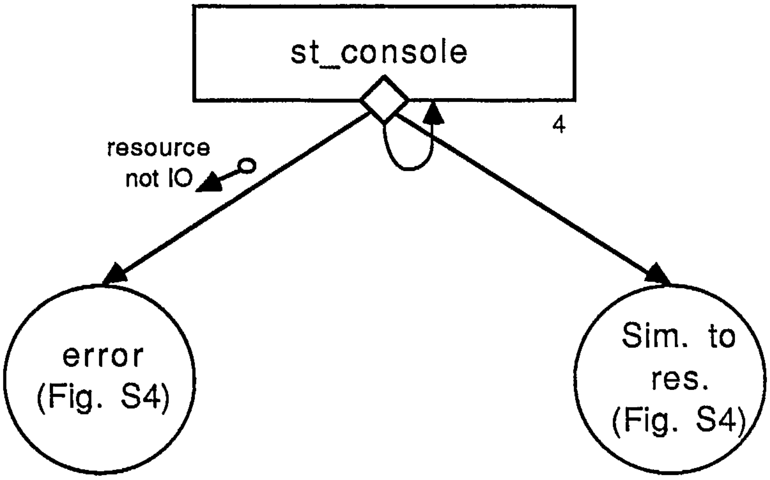


Figure S5: st_console

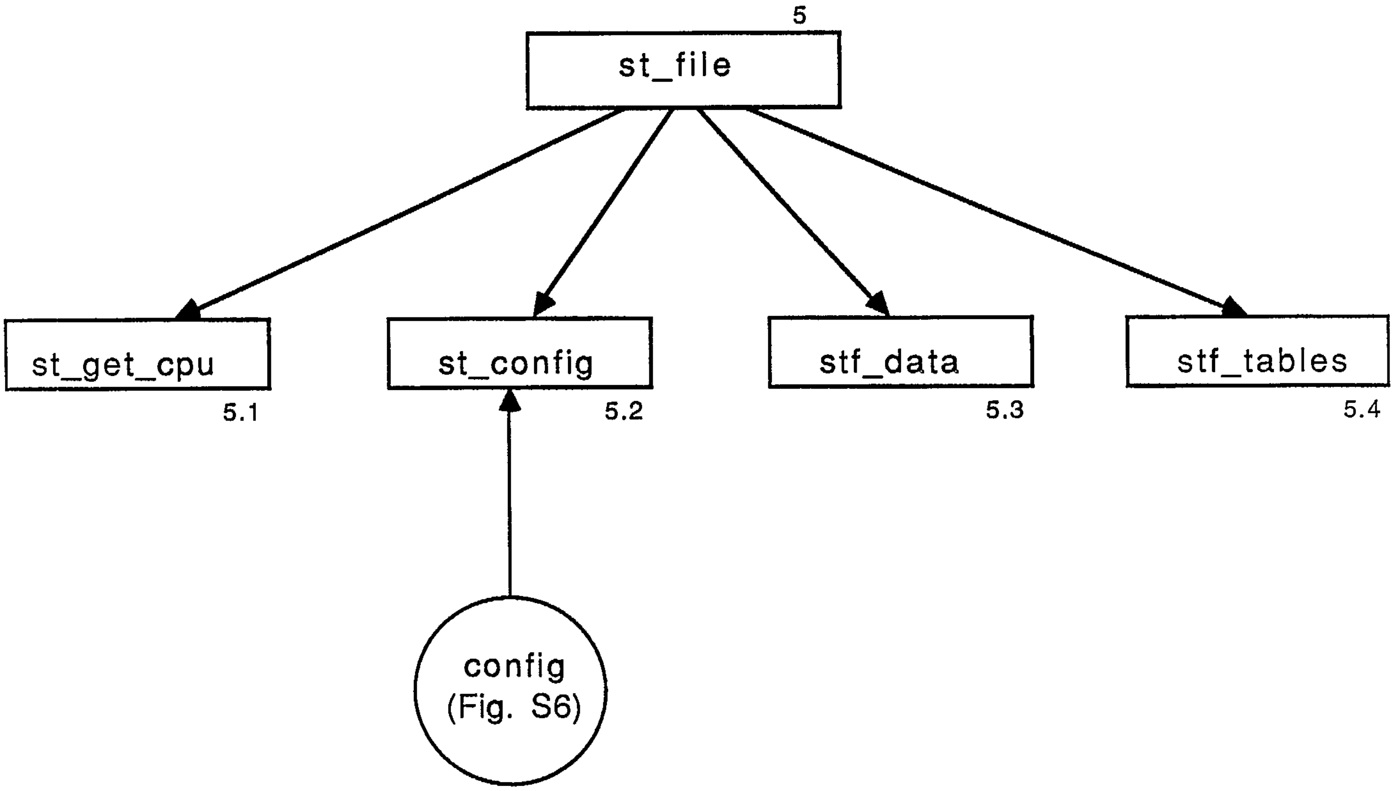


Figure S6: st_file

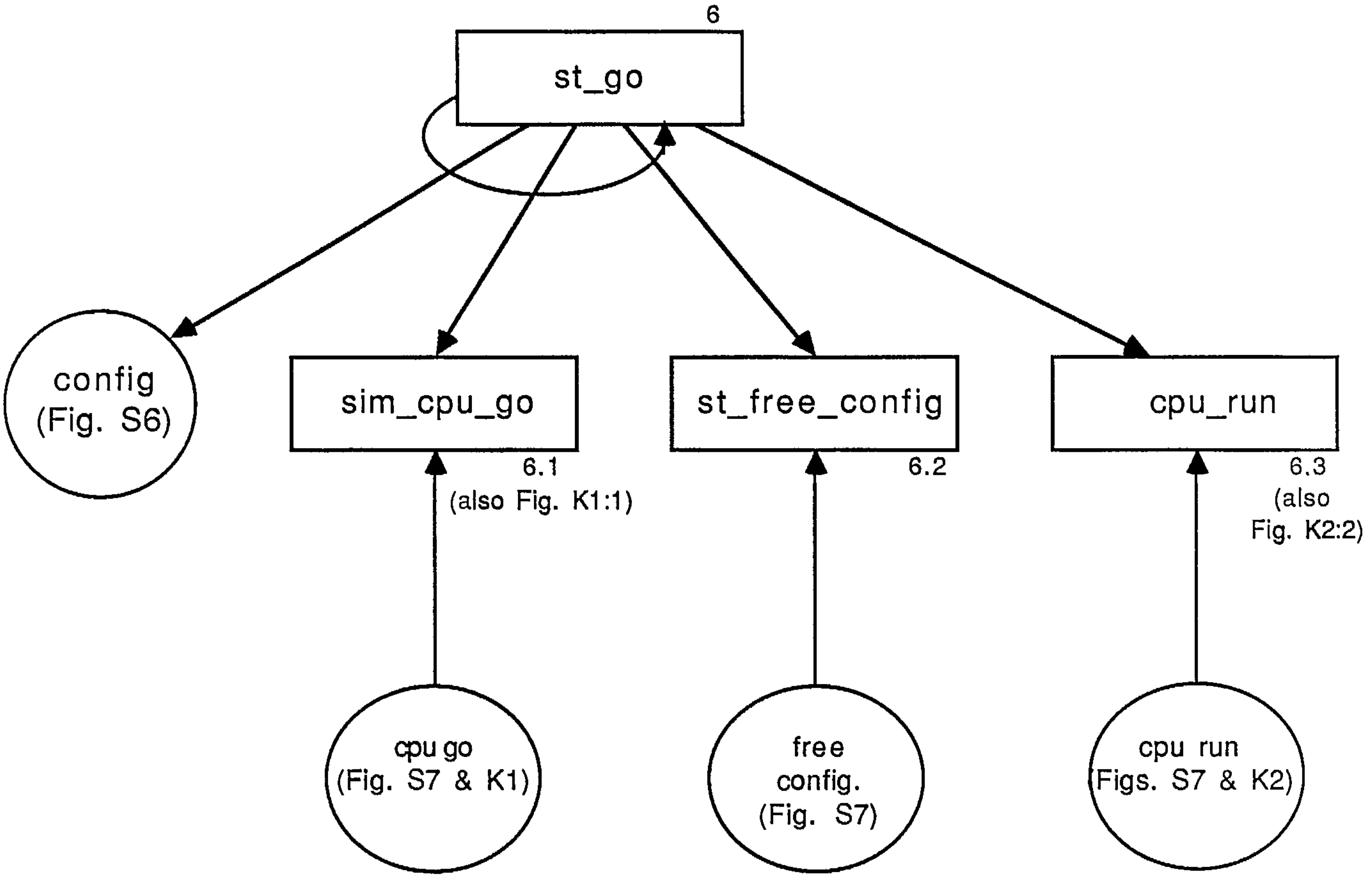


Figure S7: st_go

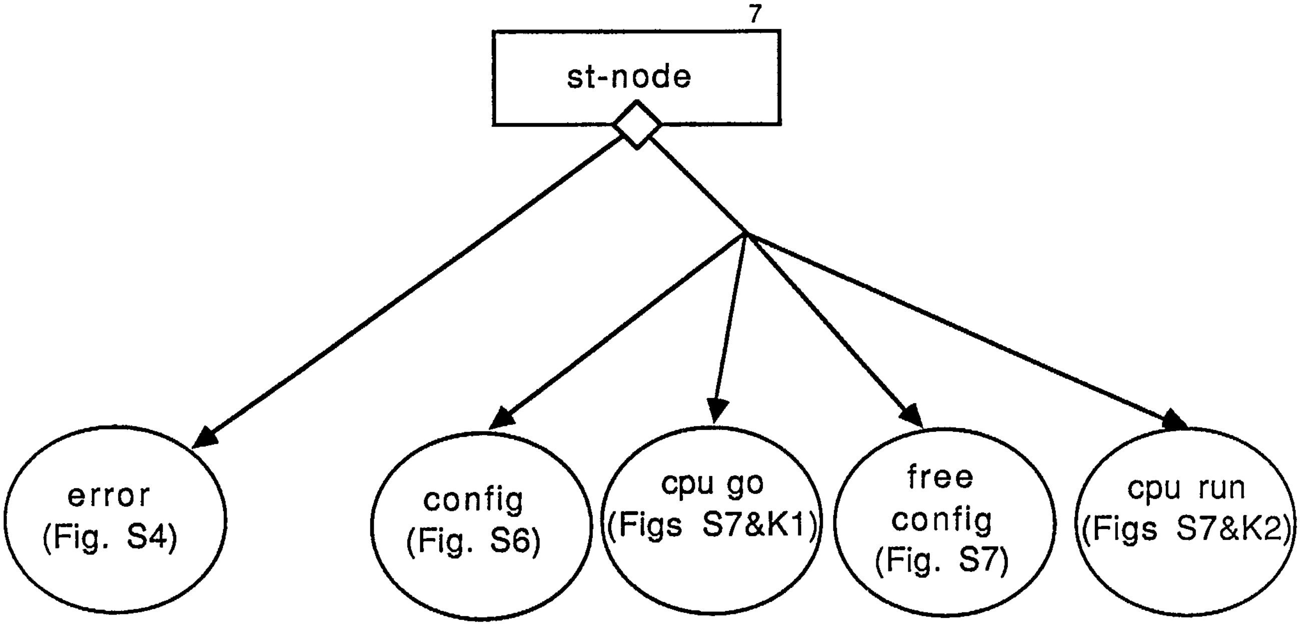


Figure S8: st_node

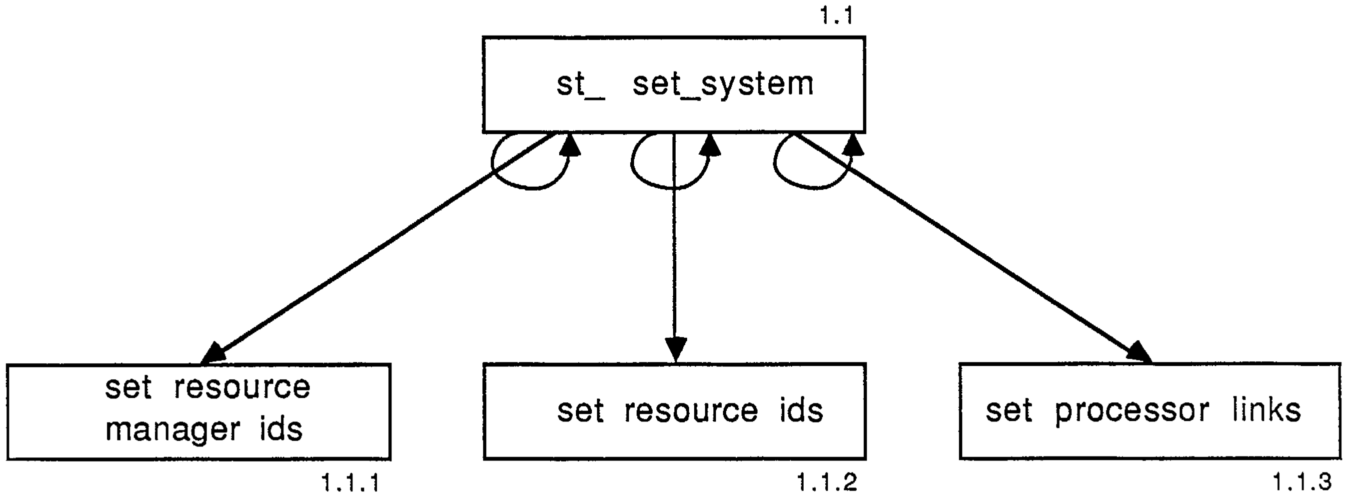


Figure S9: st_set_system

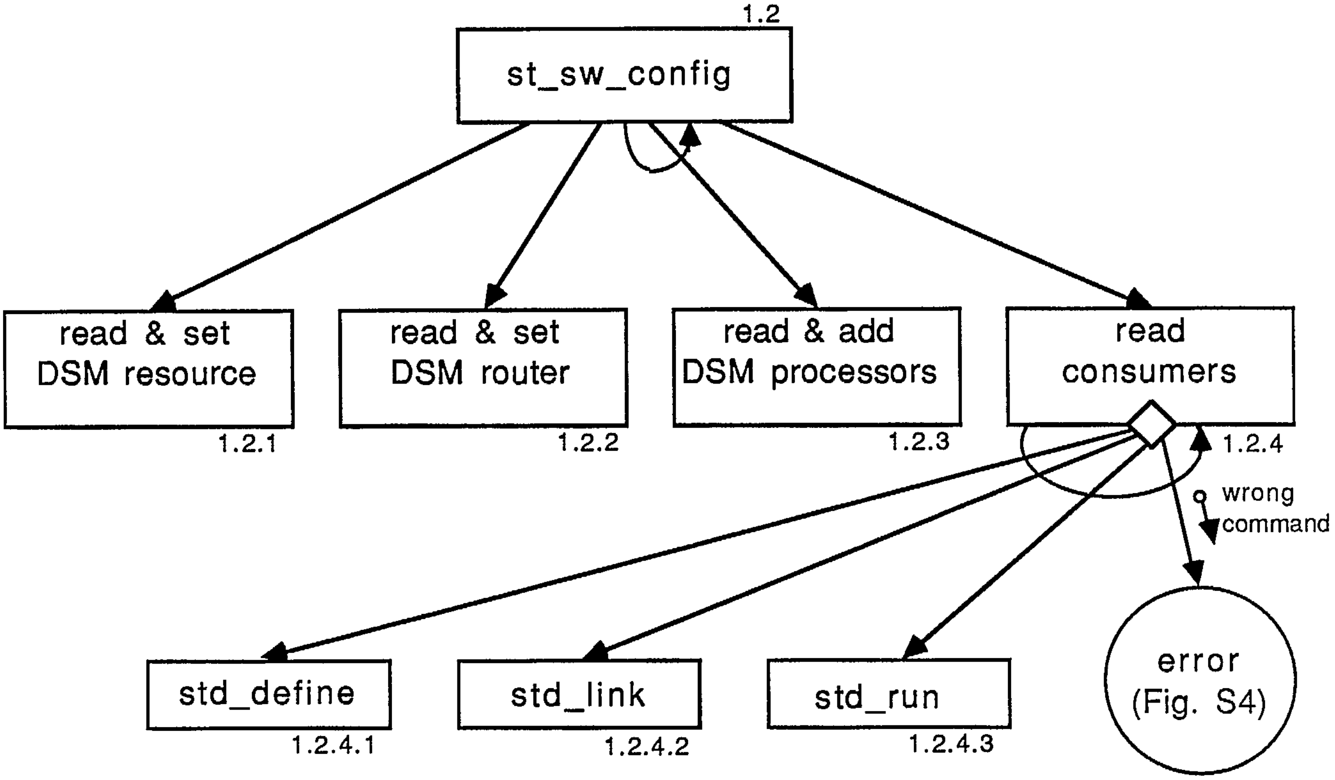


Figure S10: st_sw_config

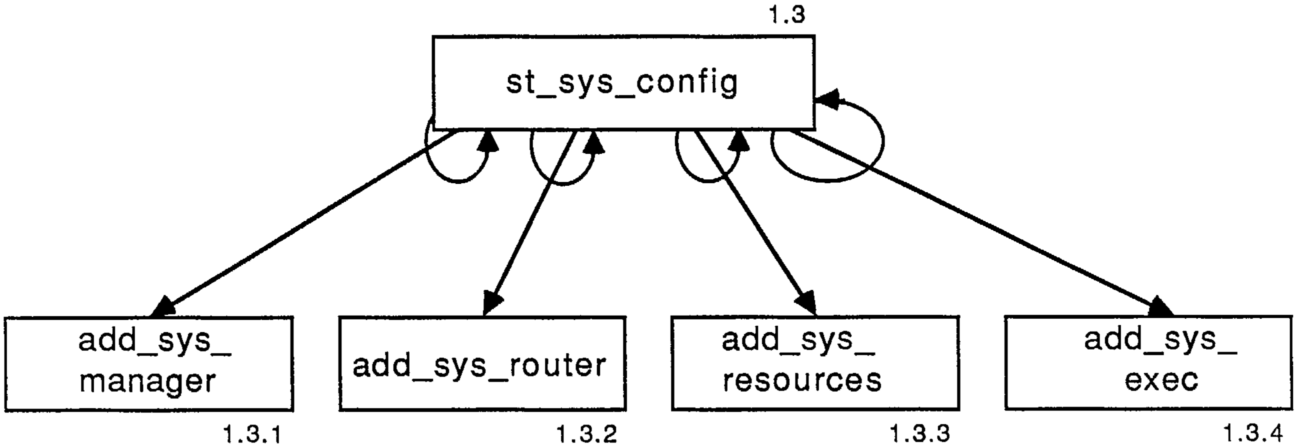


Figure S11: st_sys_config

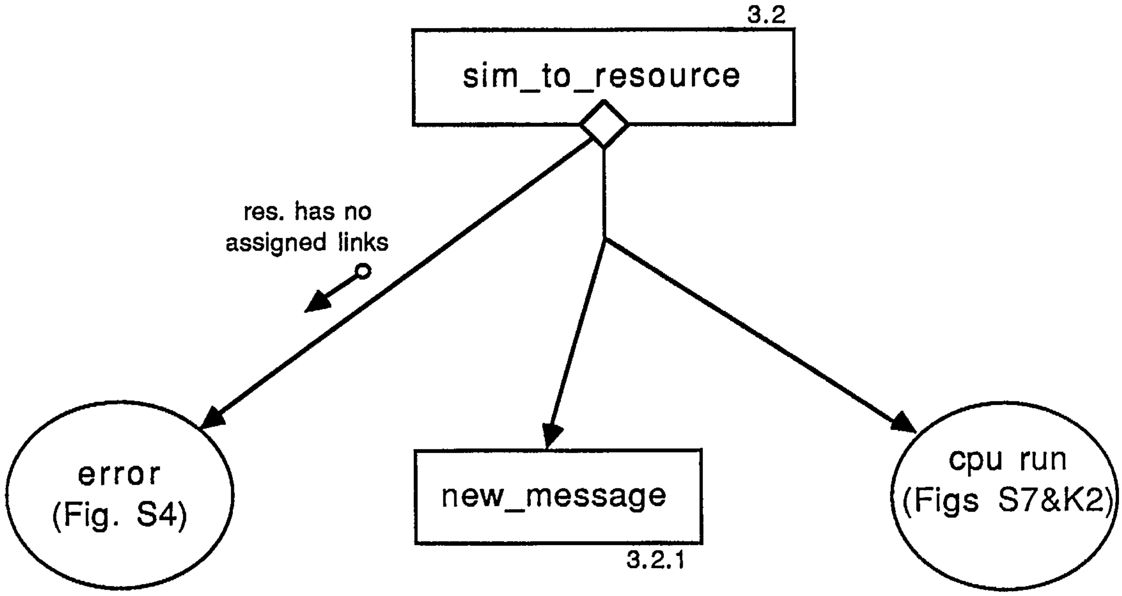


Figure S12: sim_to_resource

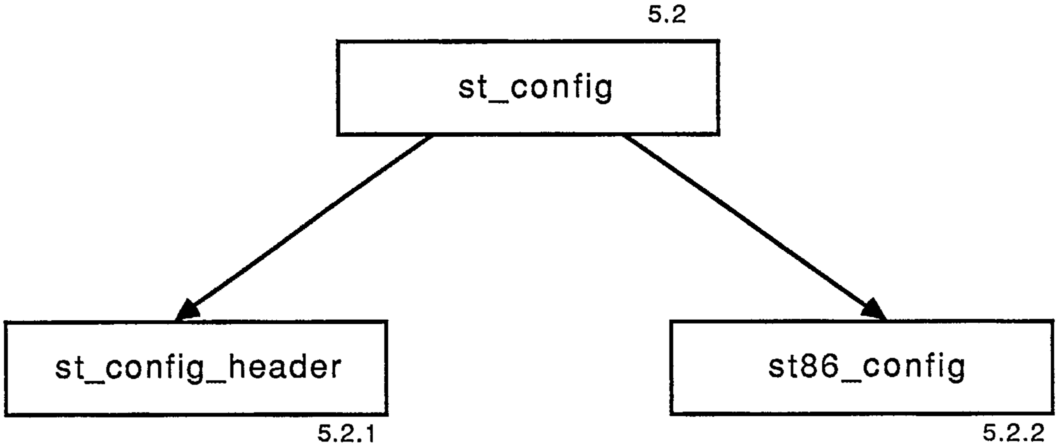


Figure S13: st_config

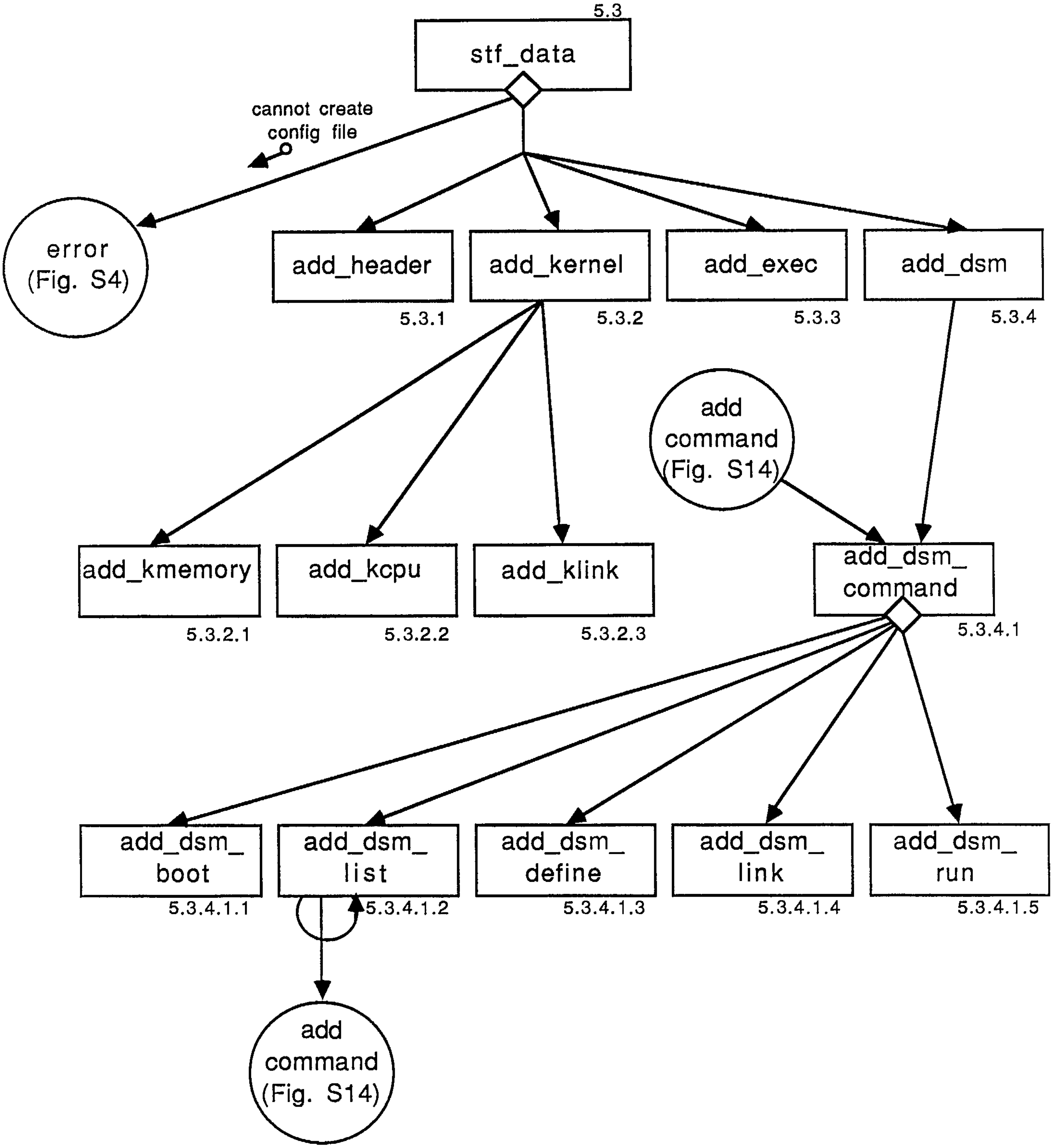


Figure S14: stf_data

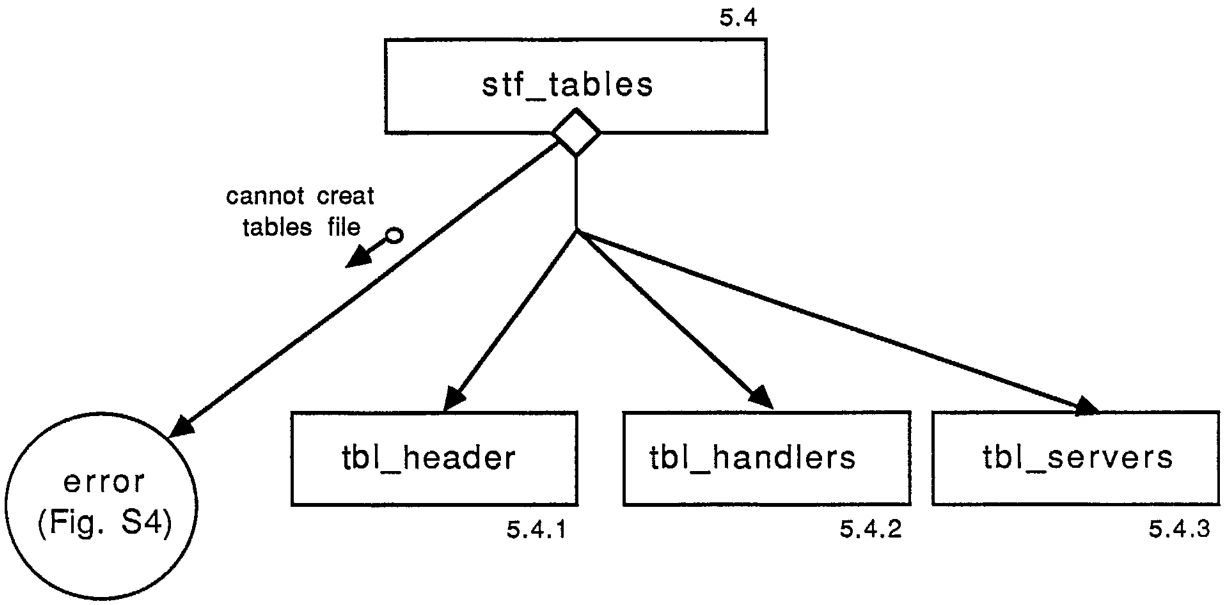


Figure S15: stf_tables

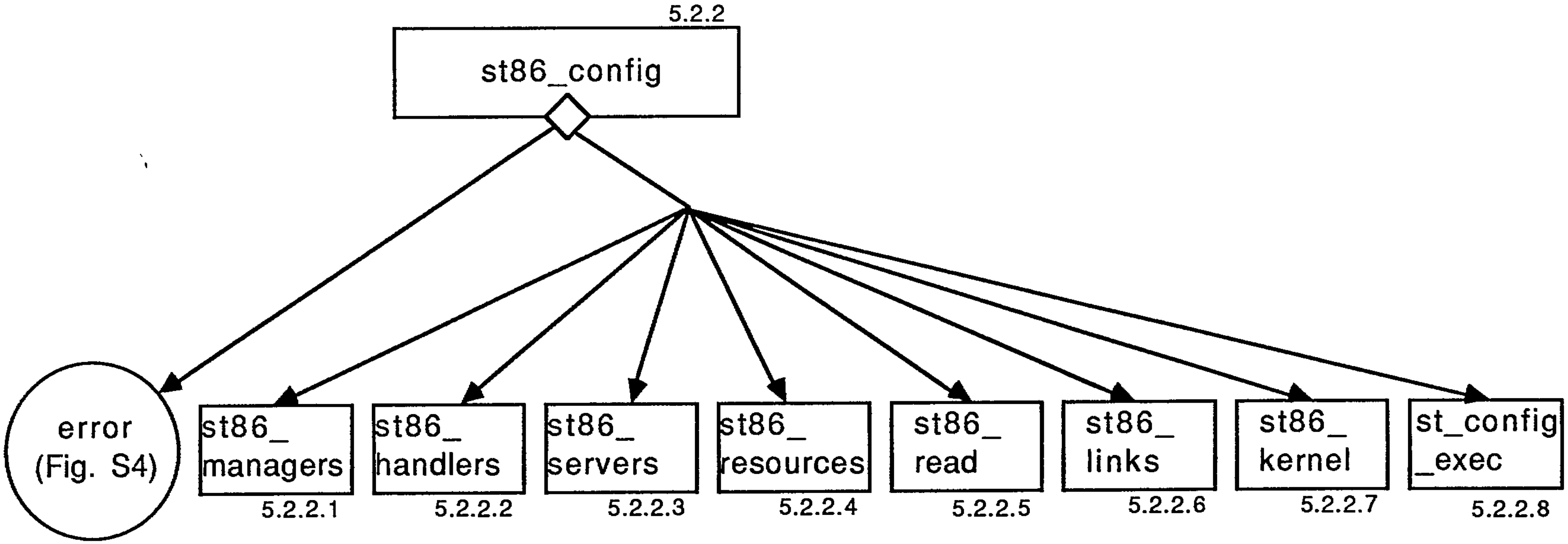


Figure S16: st86_config

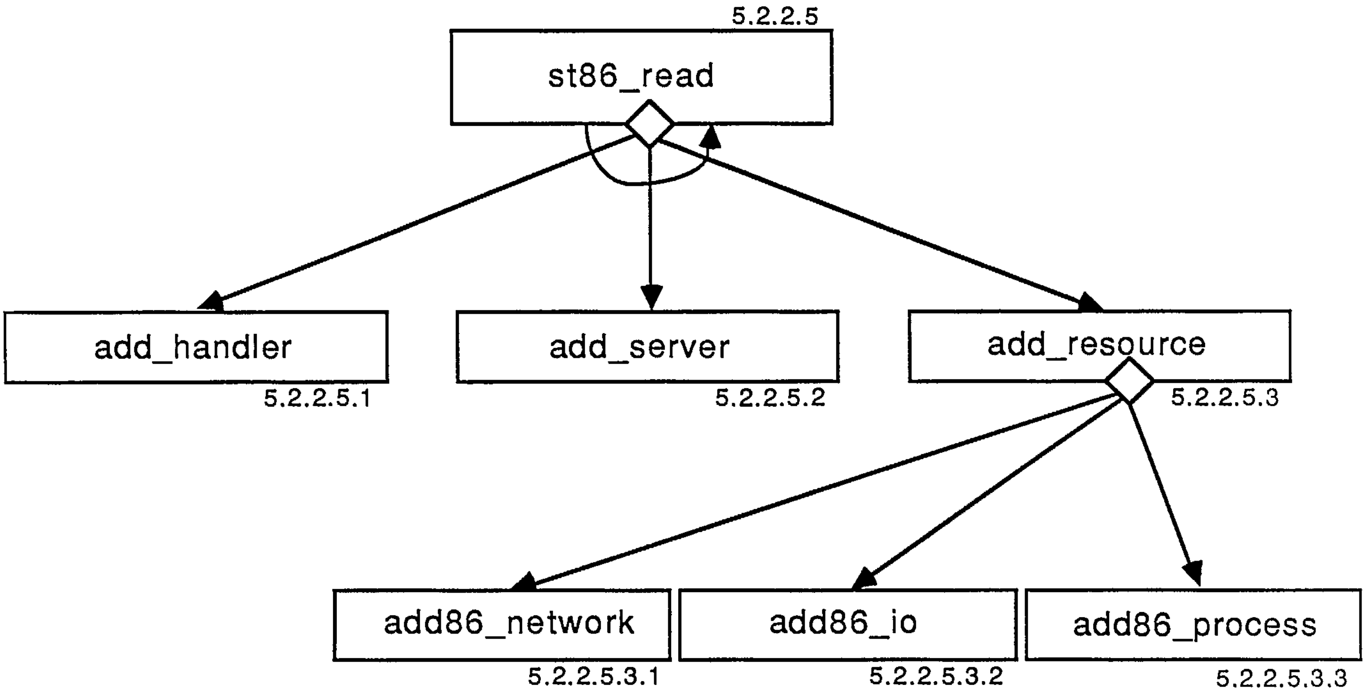


Figure S17: st86_read

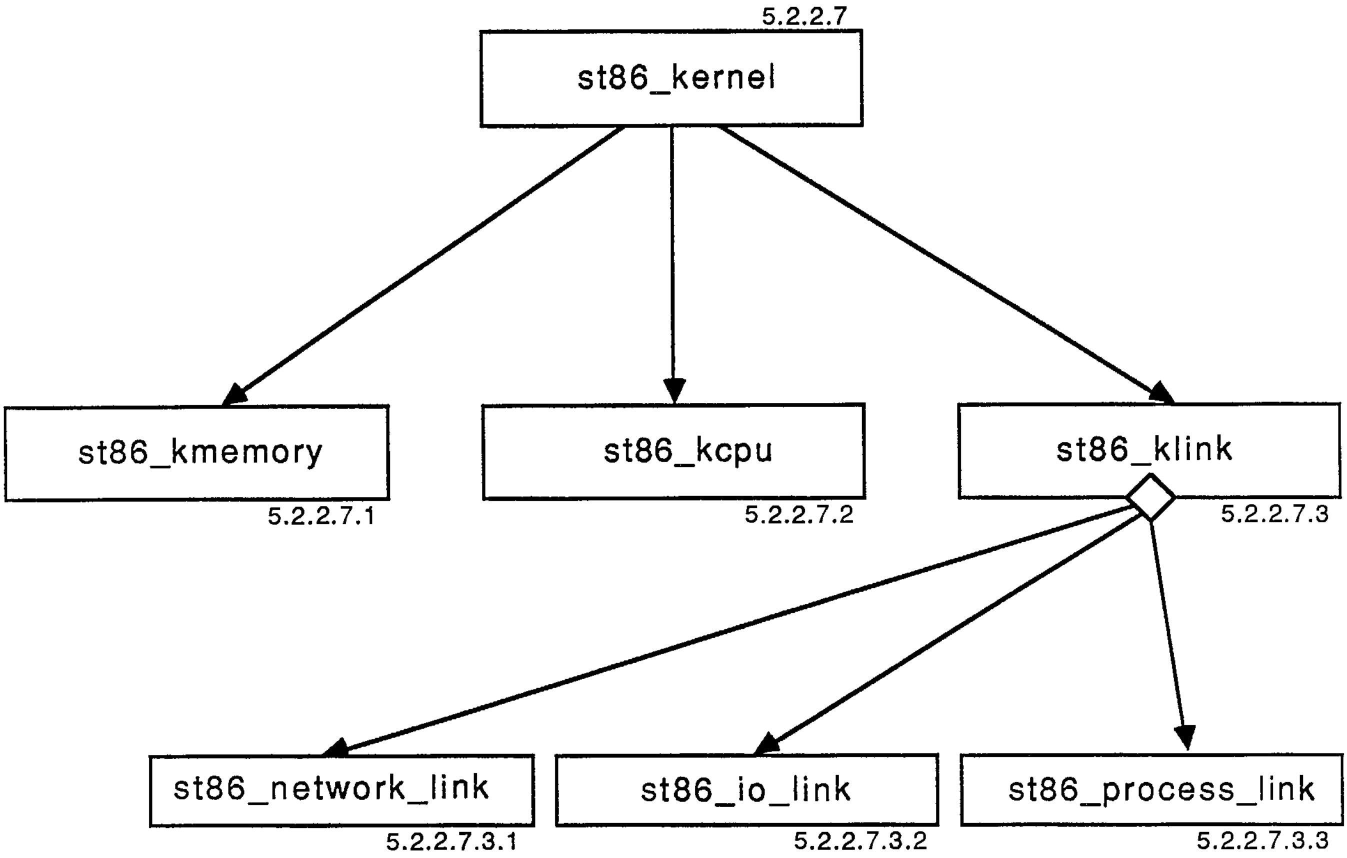


Figure S18: st86_kernel

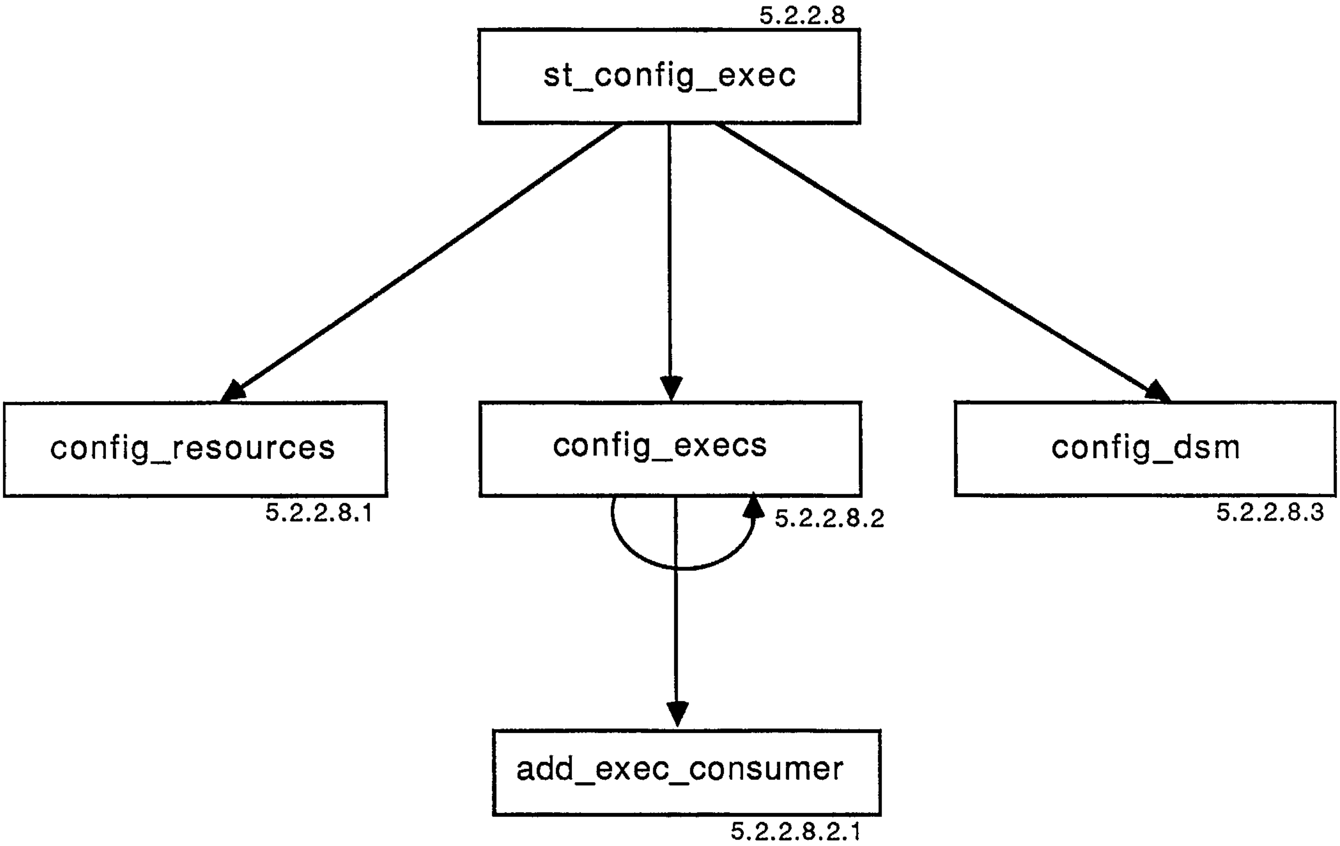


Figure S19: st_config_exec

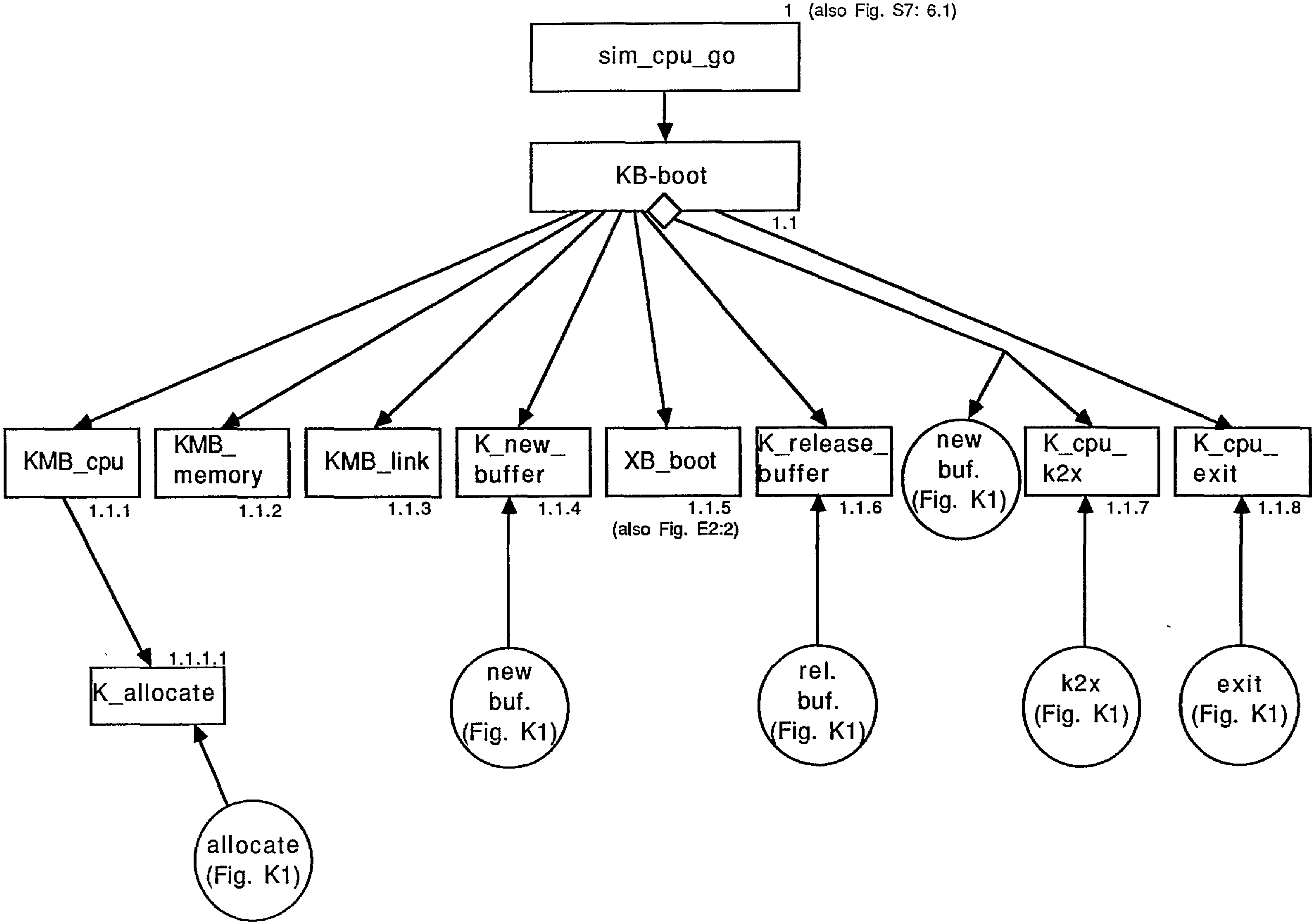


Figure K1: sim_cpu_go

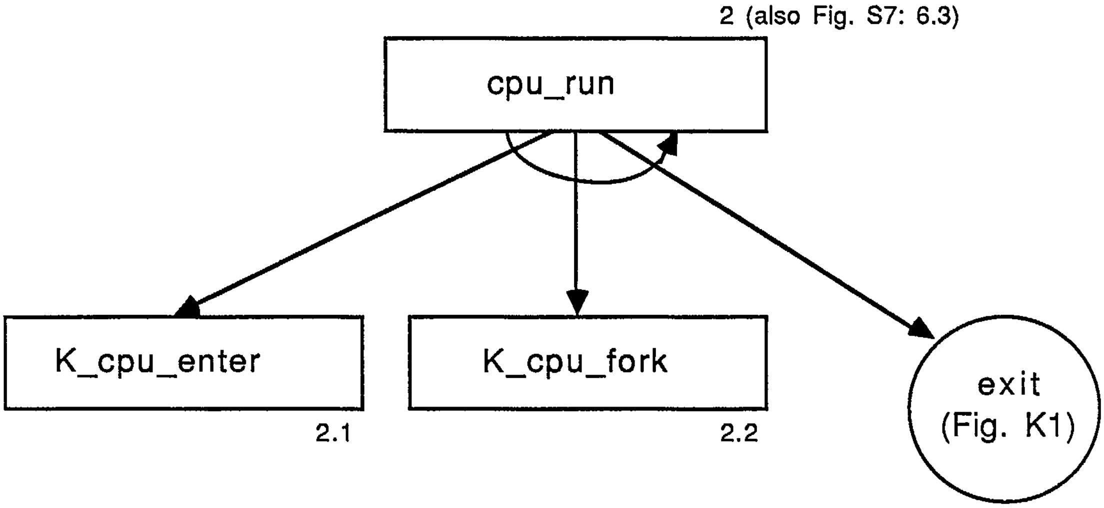


Figure K2: cpu_run

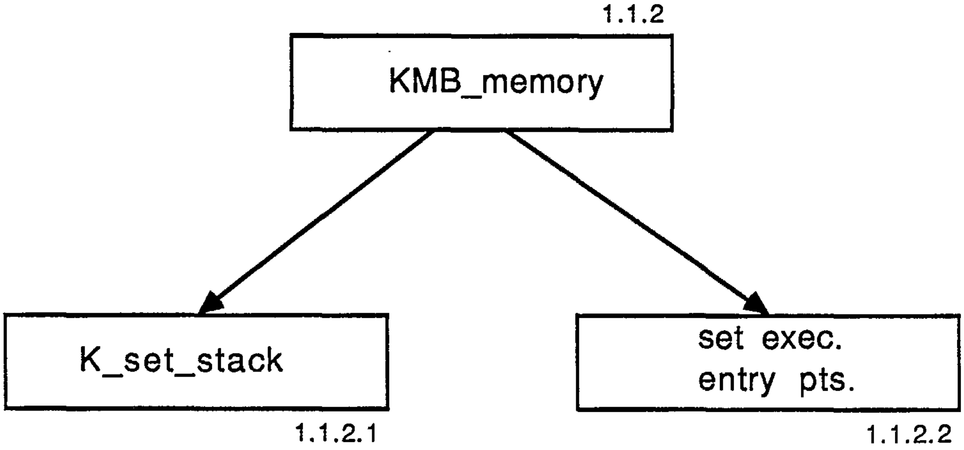


Figure K3: KMB_memory

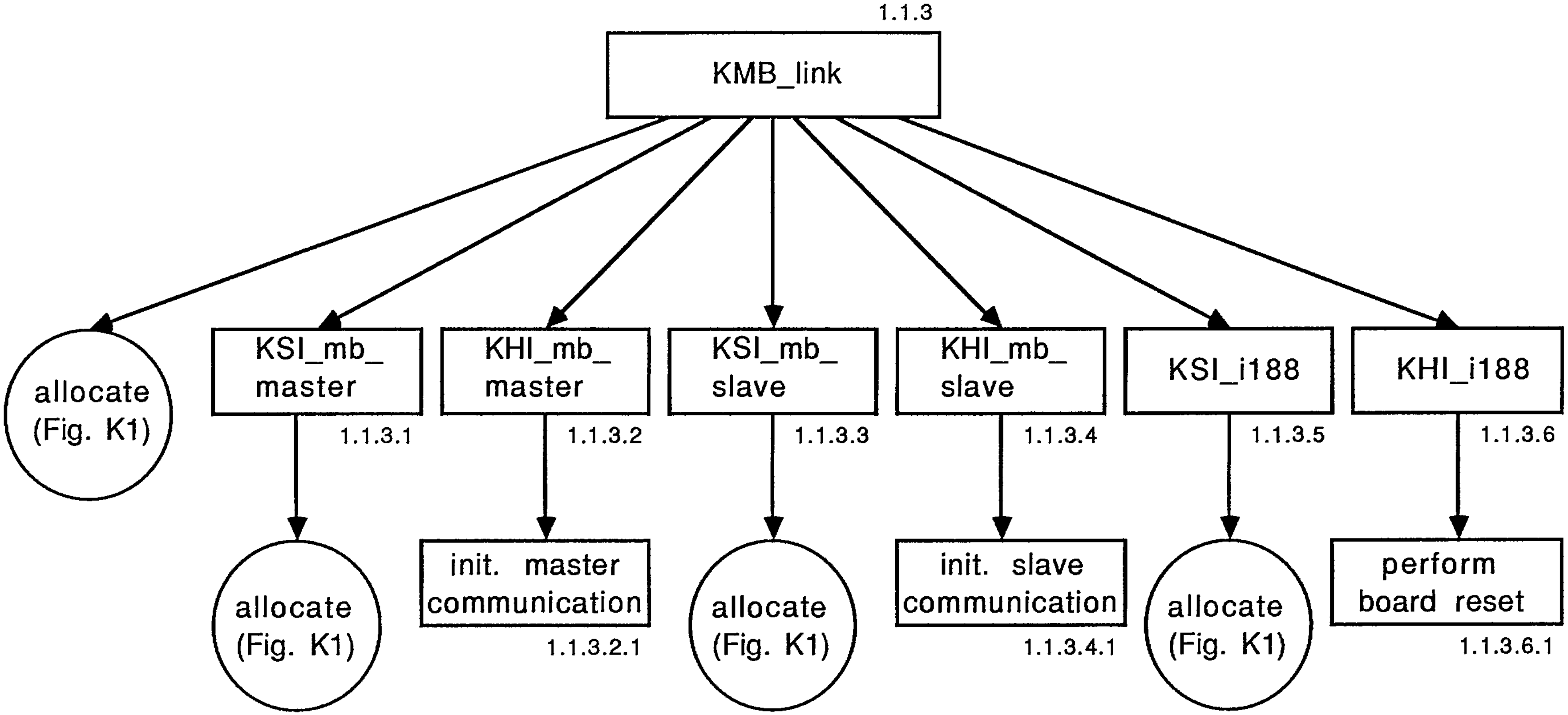


Figure K4: KMB_link

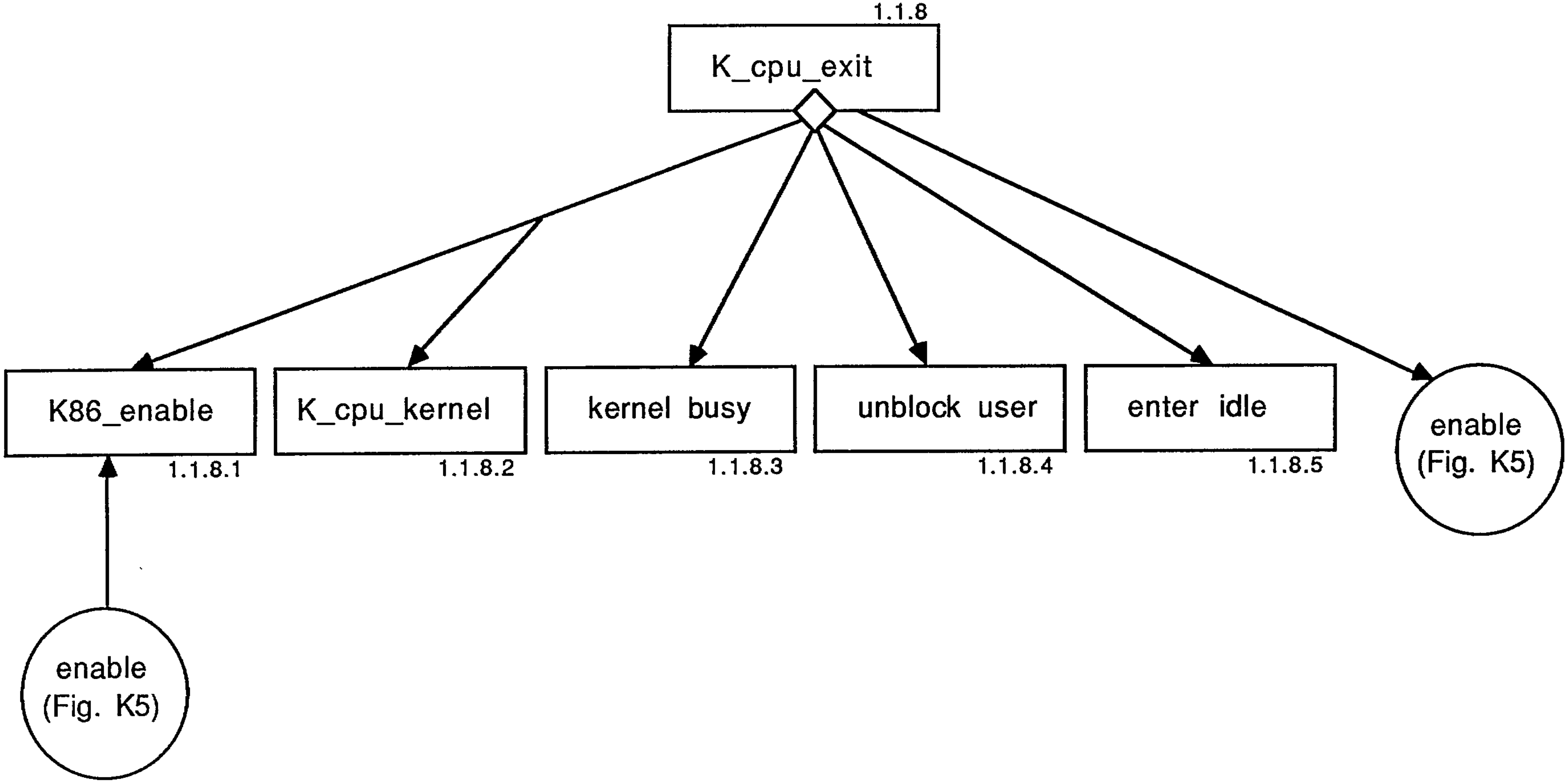


Figure K5: K_cpu_exit

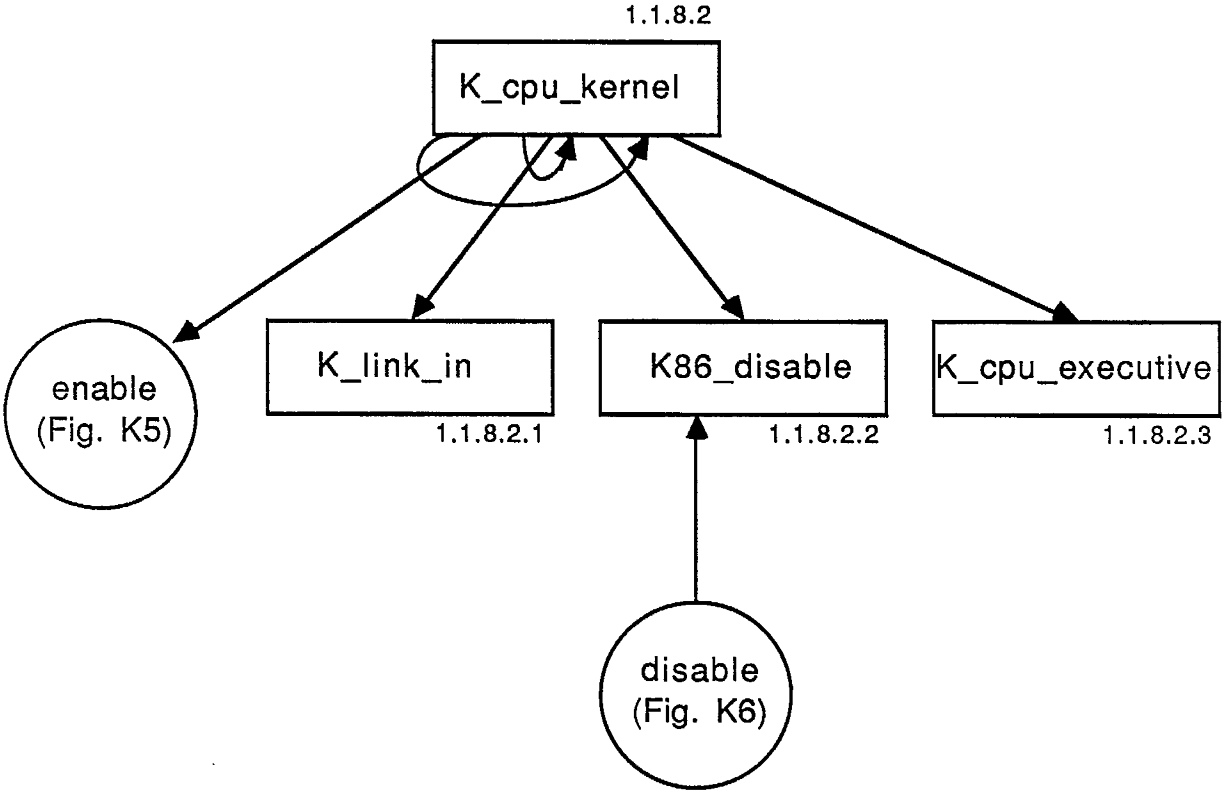


Figure K6: K_cpu_kernel

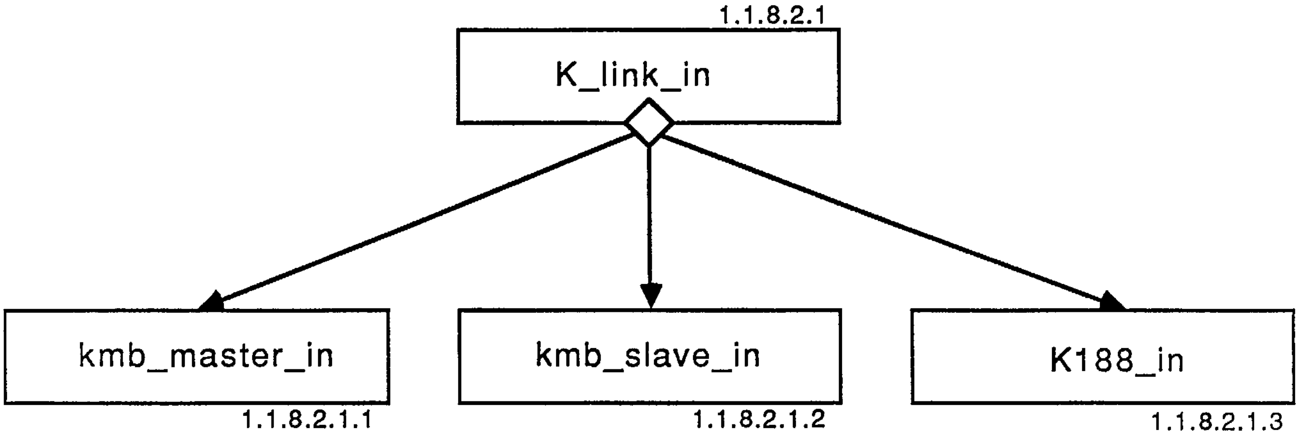


Figure K7: K_link_in

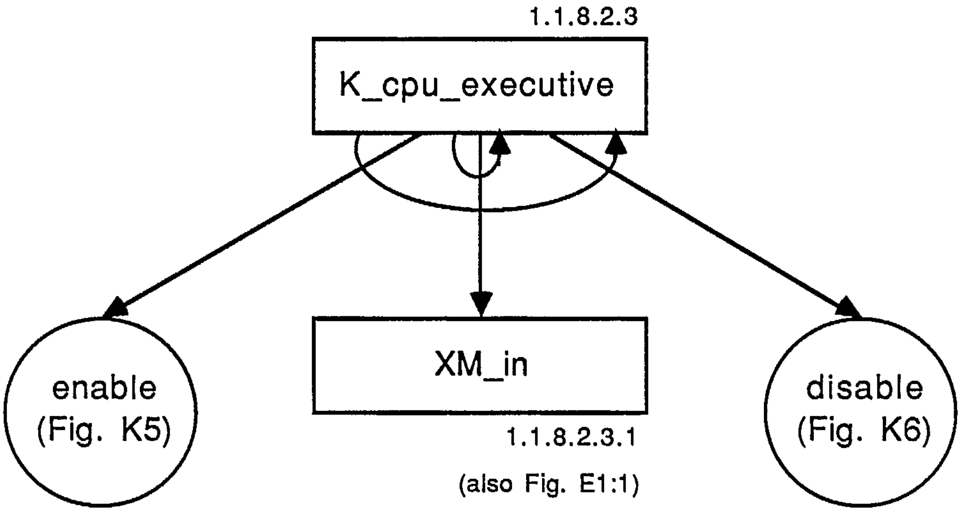


Figure K8: K_cpu_executive

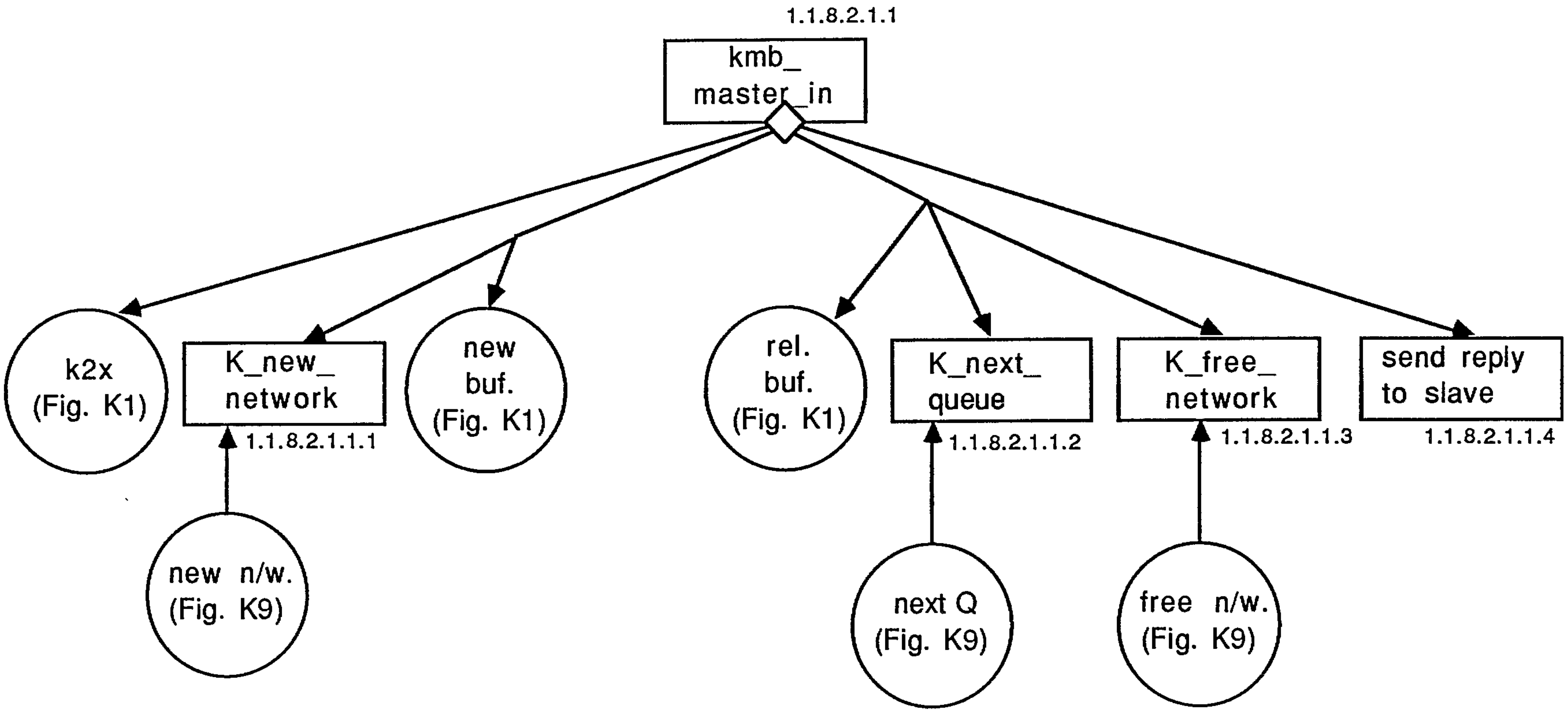


Figure K9: kmb_master_in

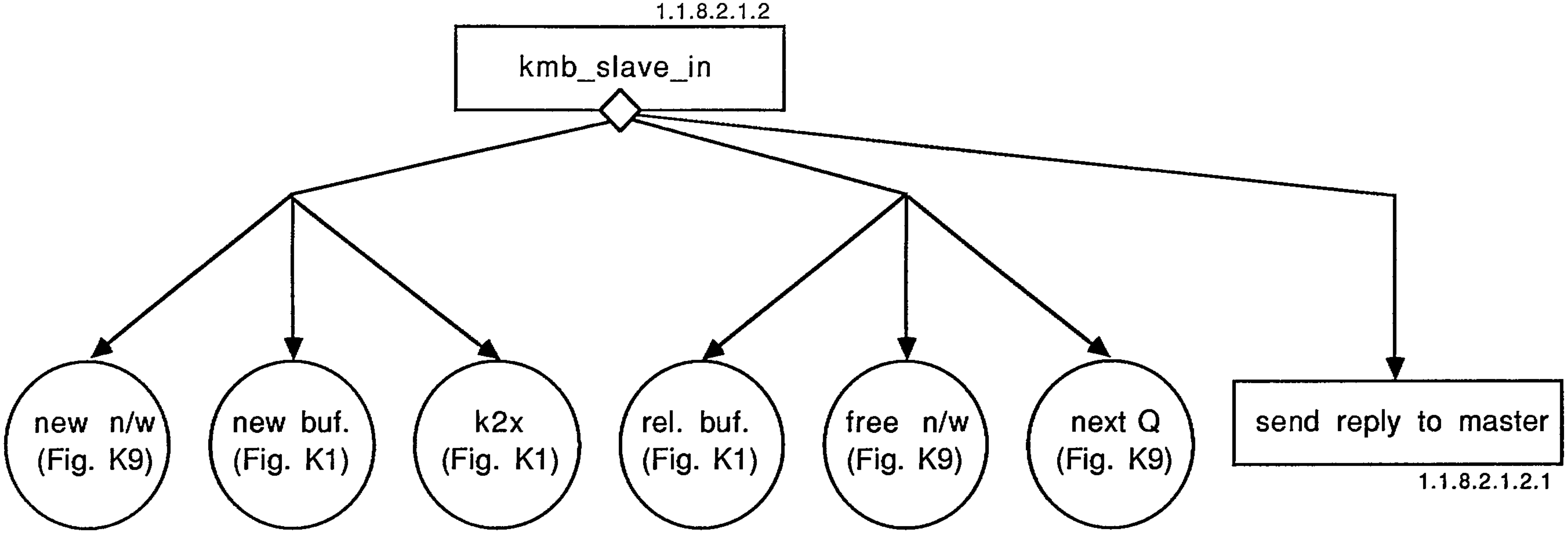


Figure K10: kmb_slave_in

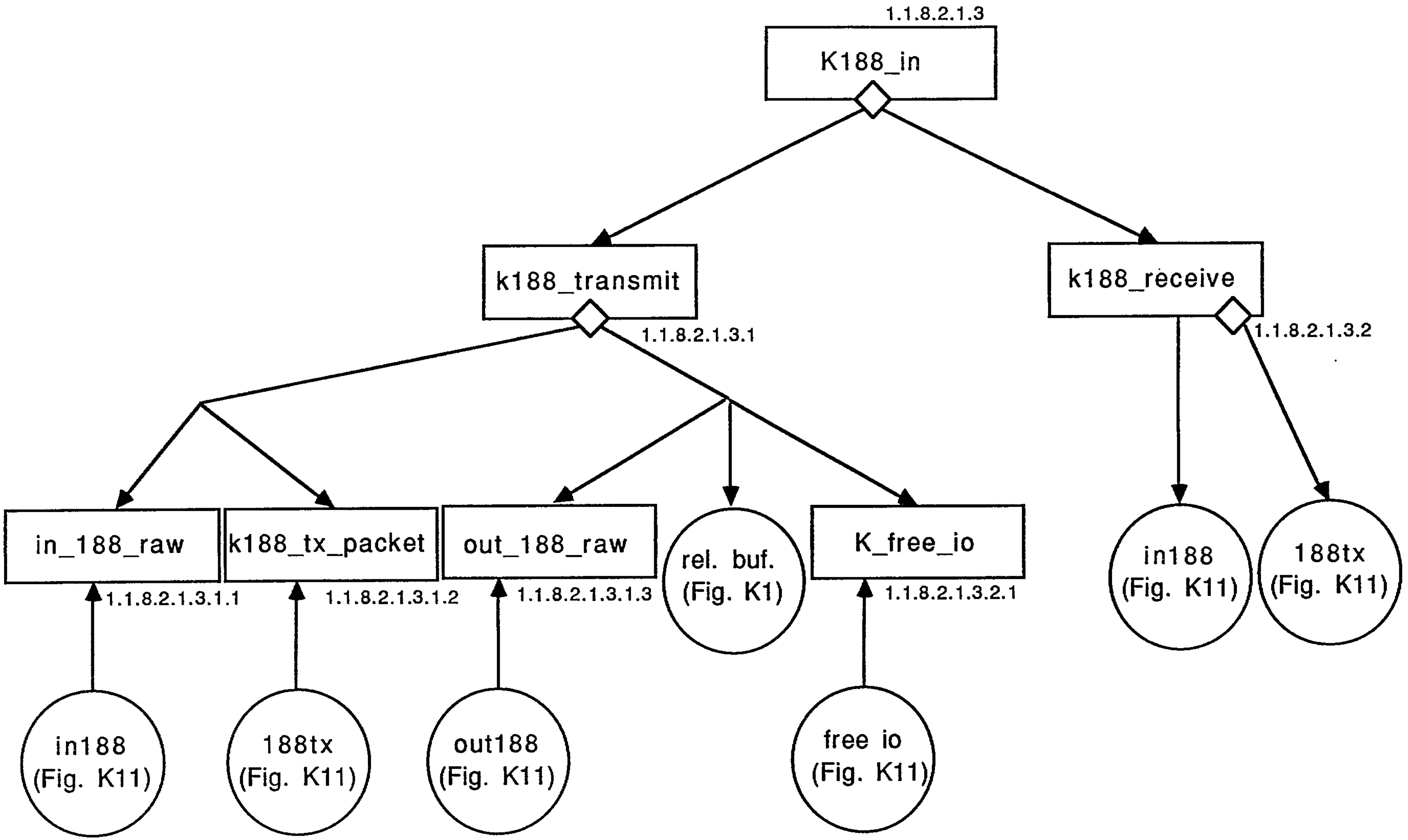


Figure K11: K188_in

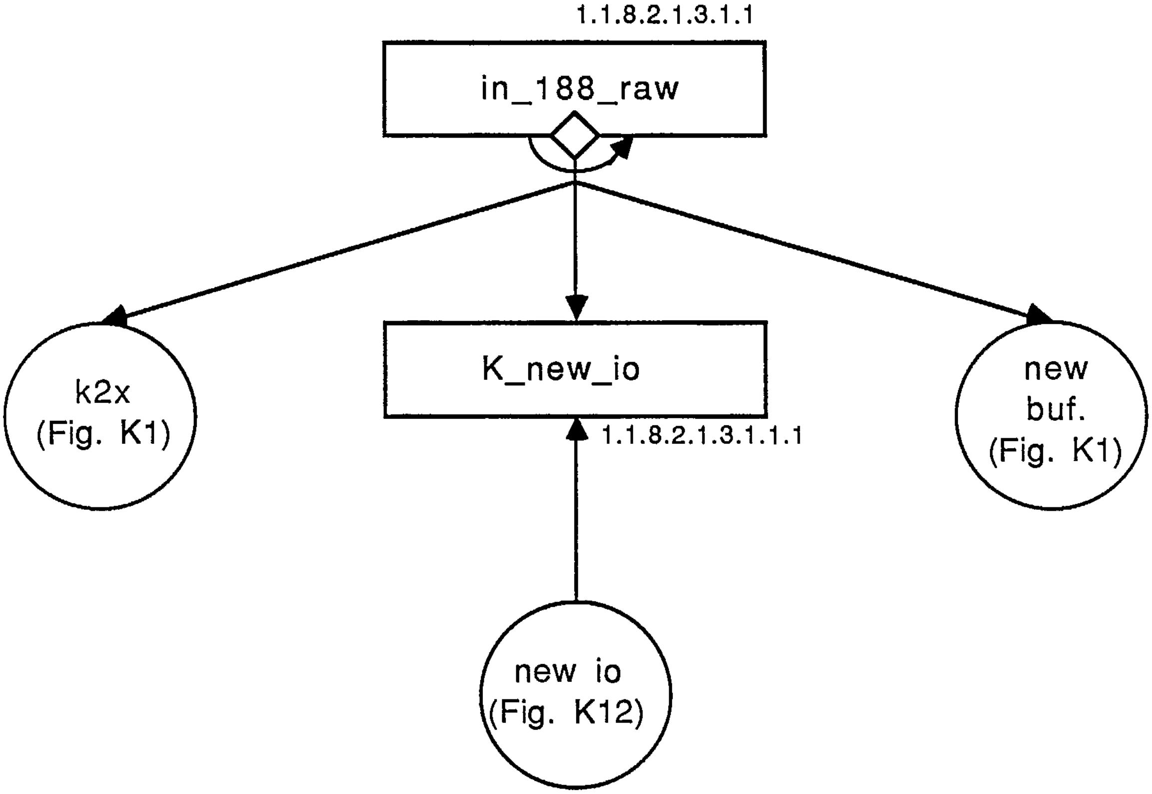


Figure K12: in_188_raw

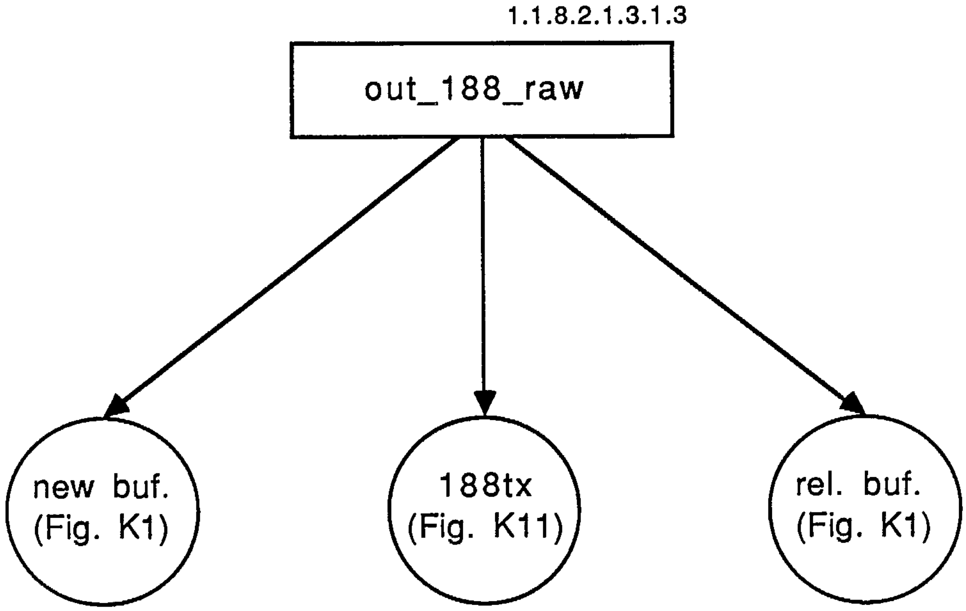


Figure K13: out_188_raw

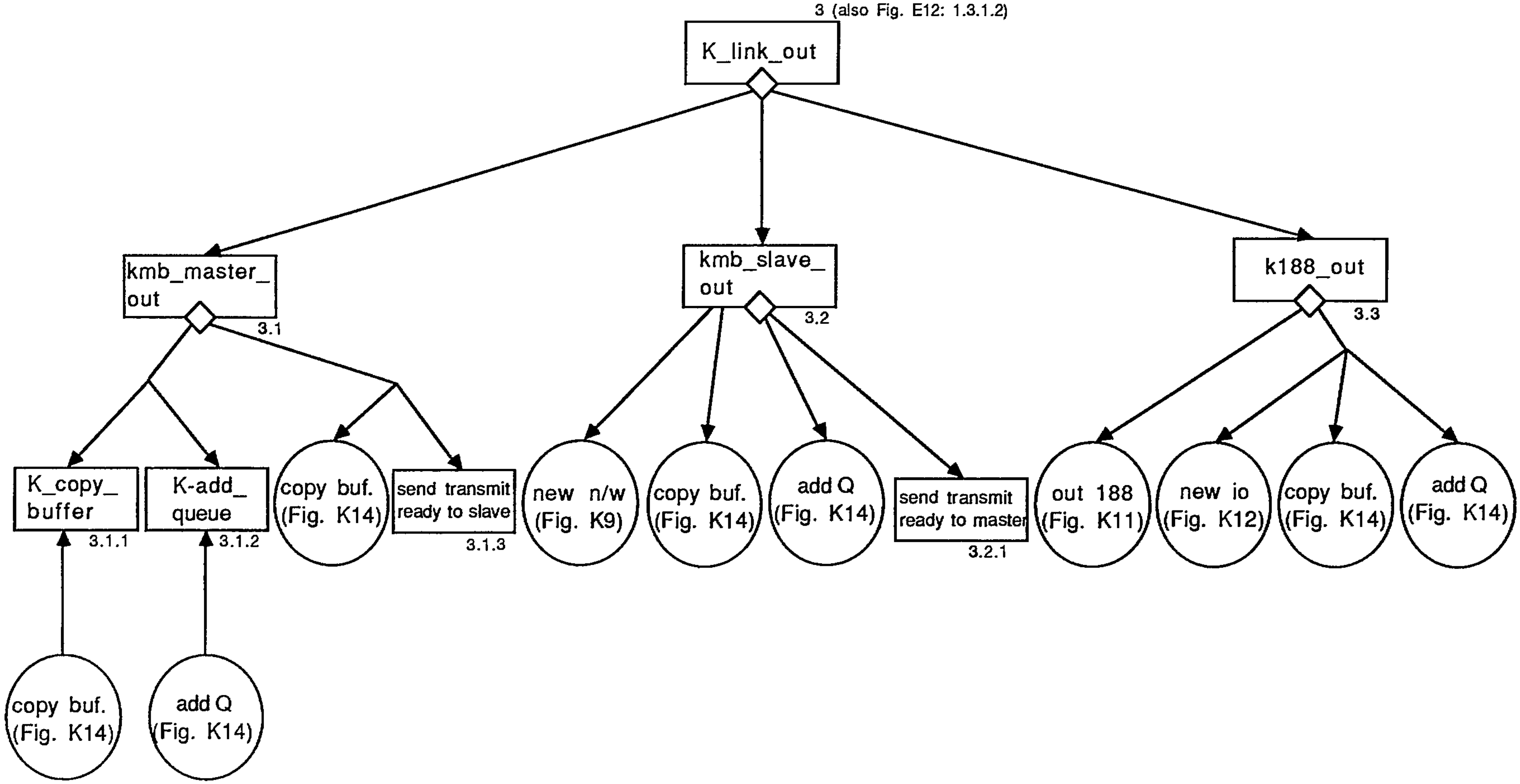


Figure K14: K_link_out

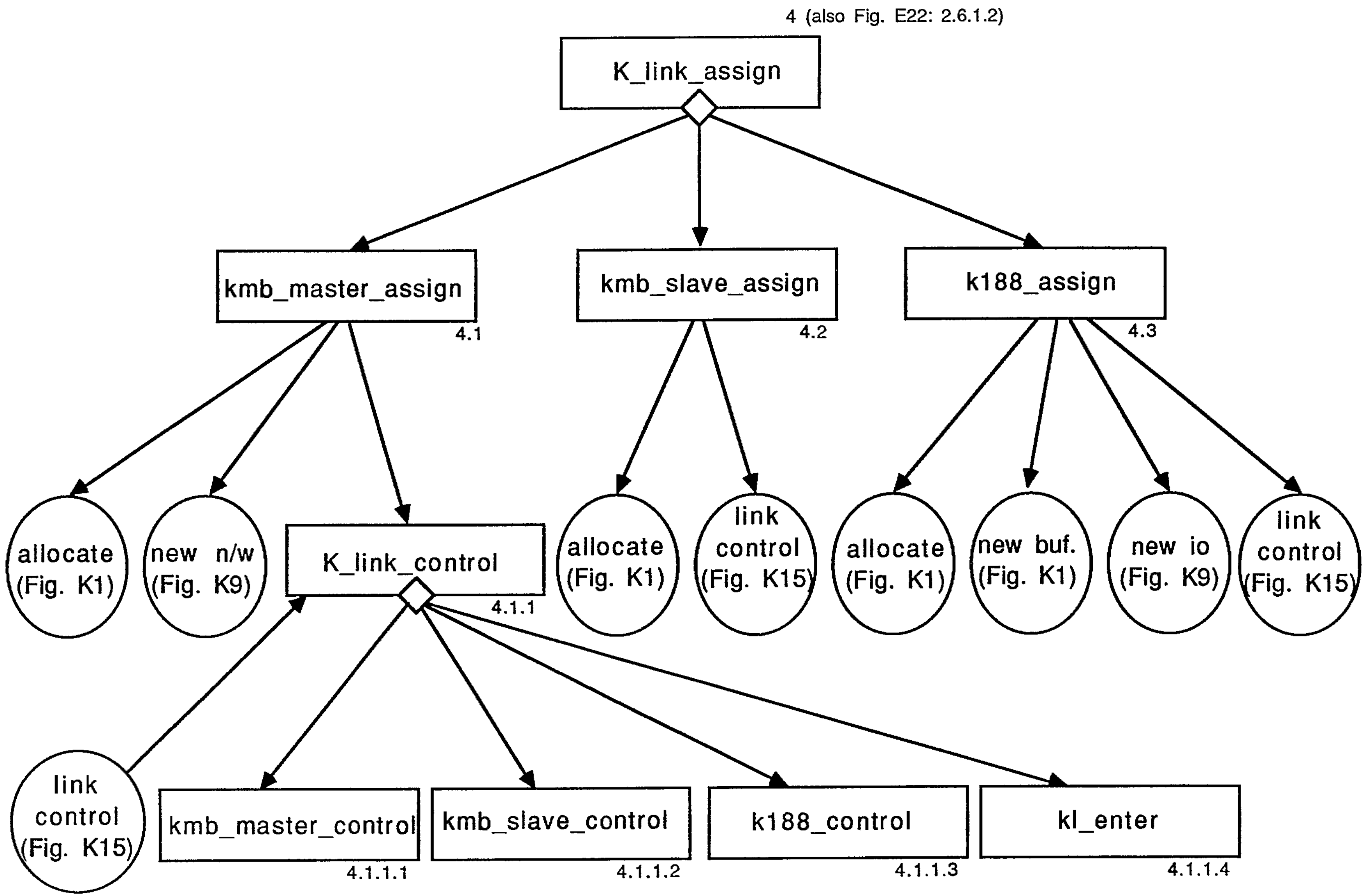


Figure K15: K_link_assign

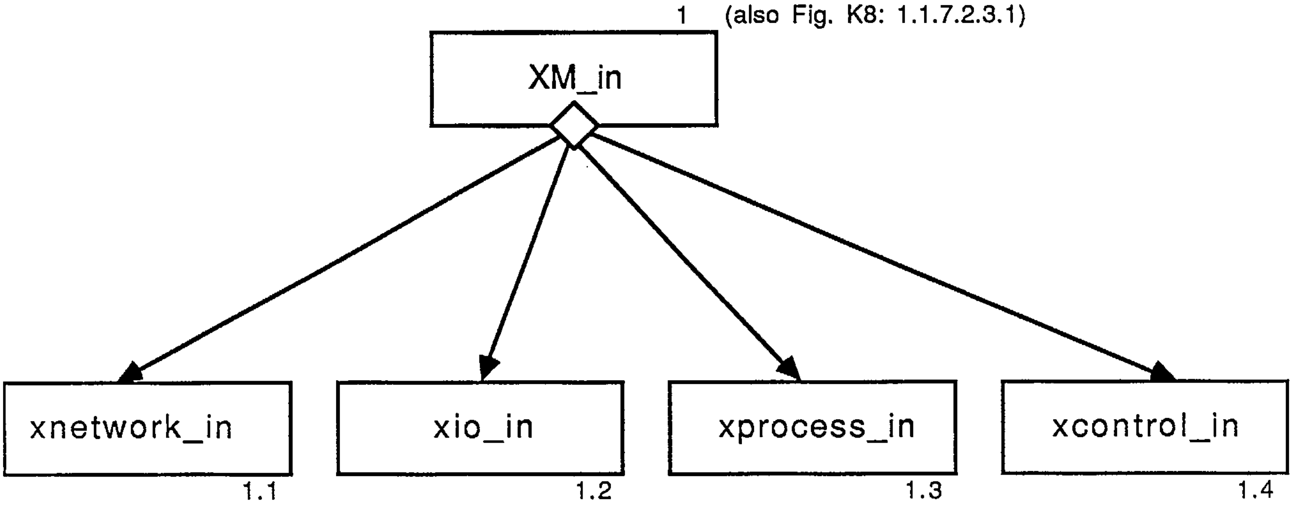


Figure E1: XM_in

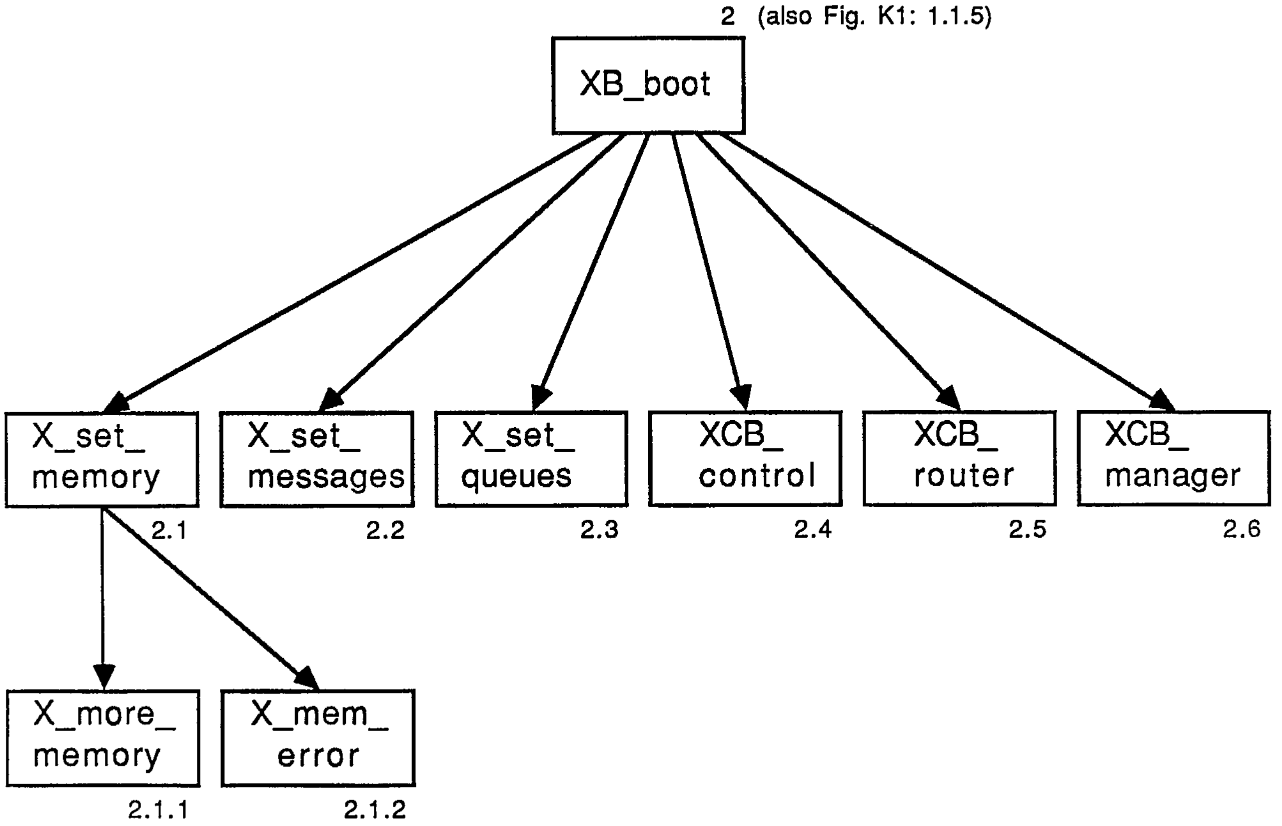


Figure E2: XB_boot

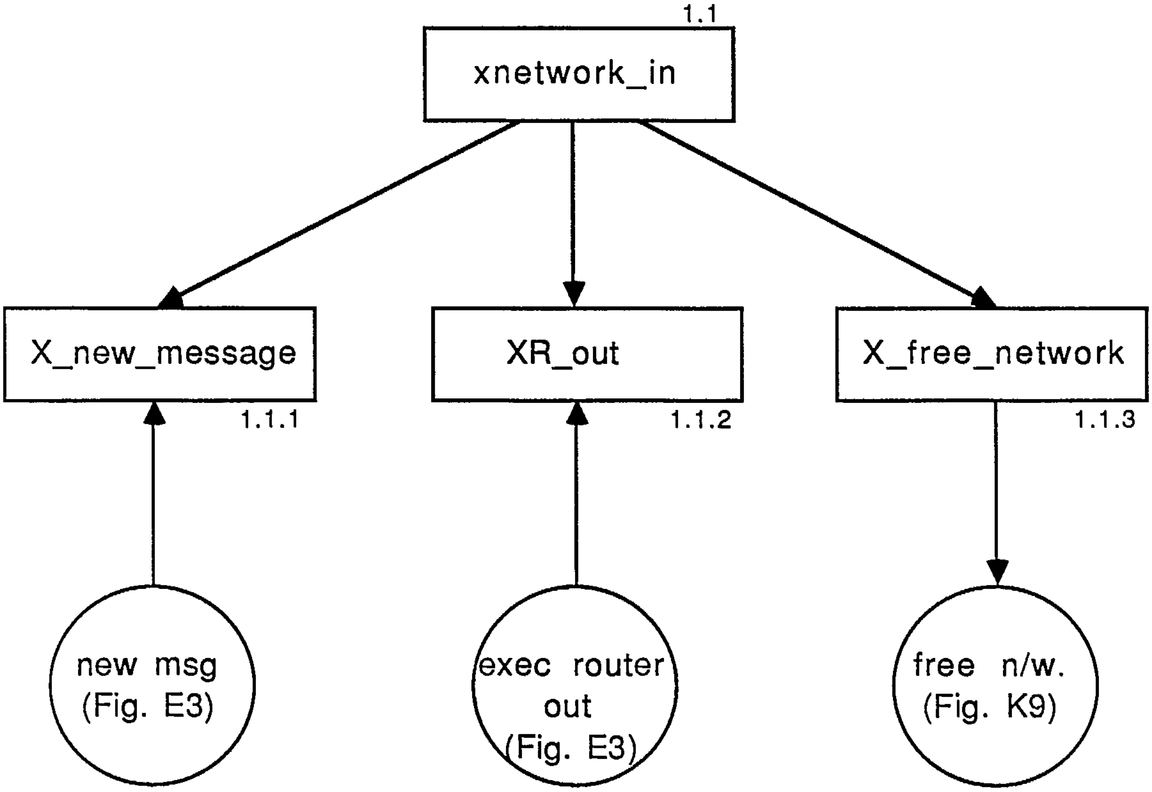


Figure E3: xnetwork_in

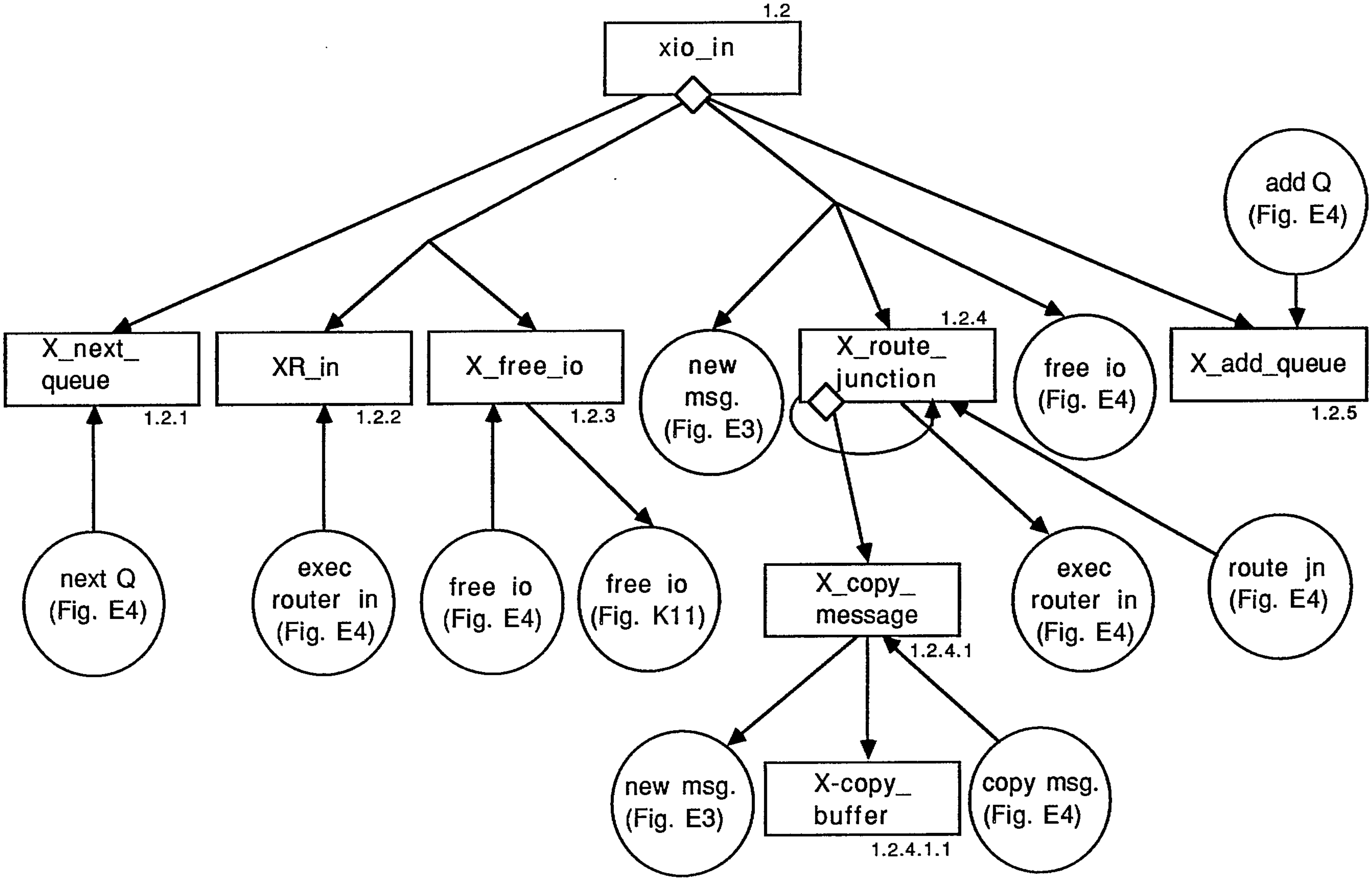


Figure E4: xio_in

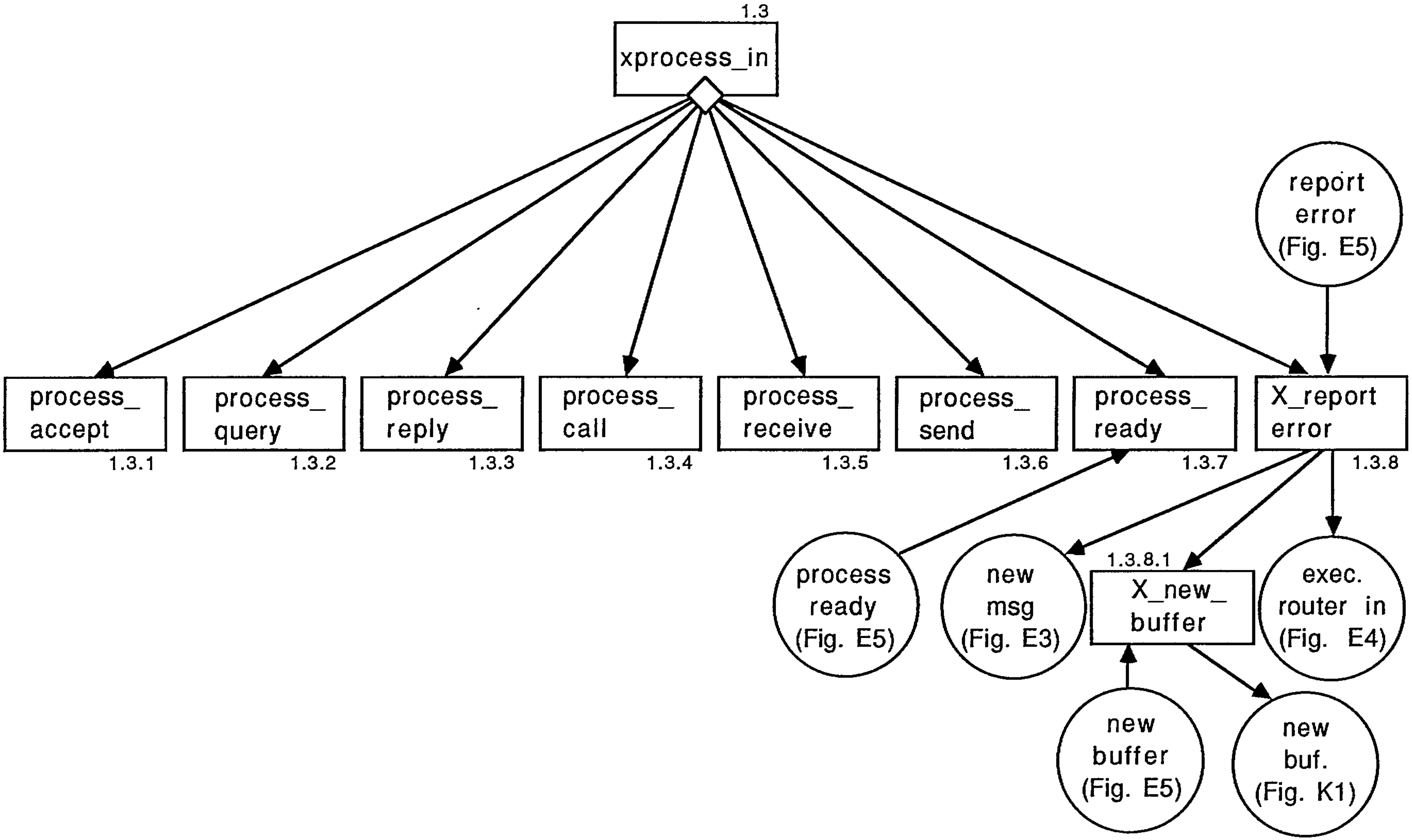


Figure E5: xprocess_in

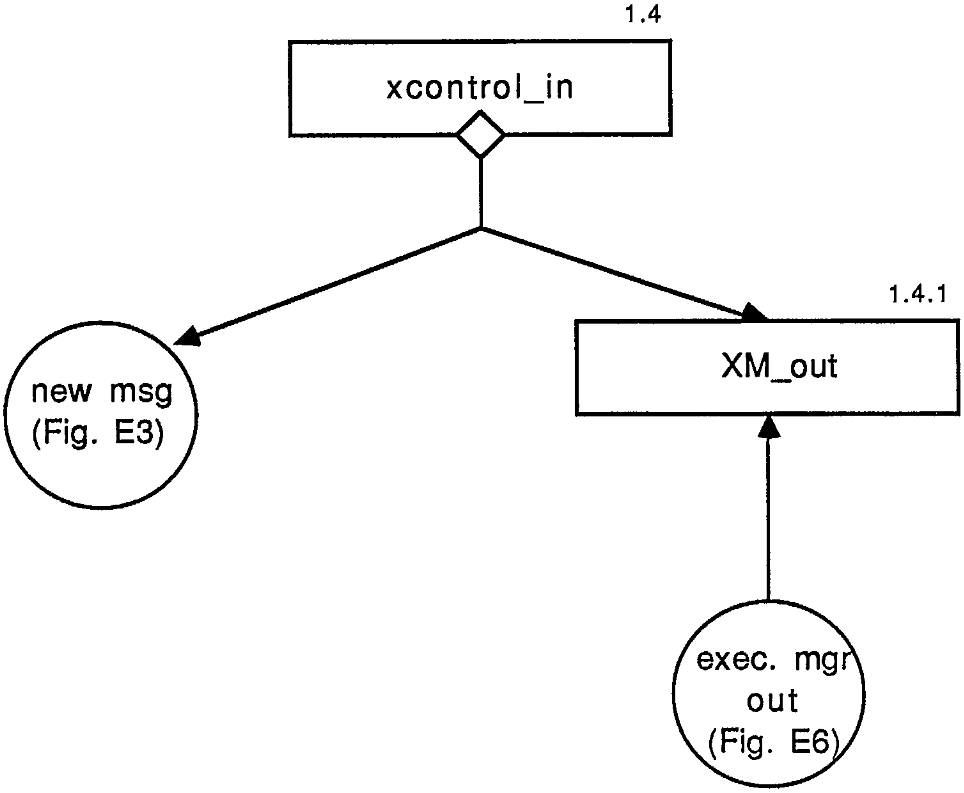


Figure E6: xcontrol_in

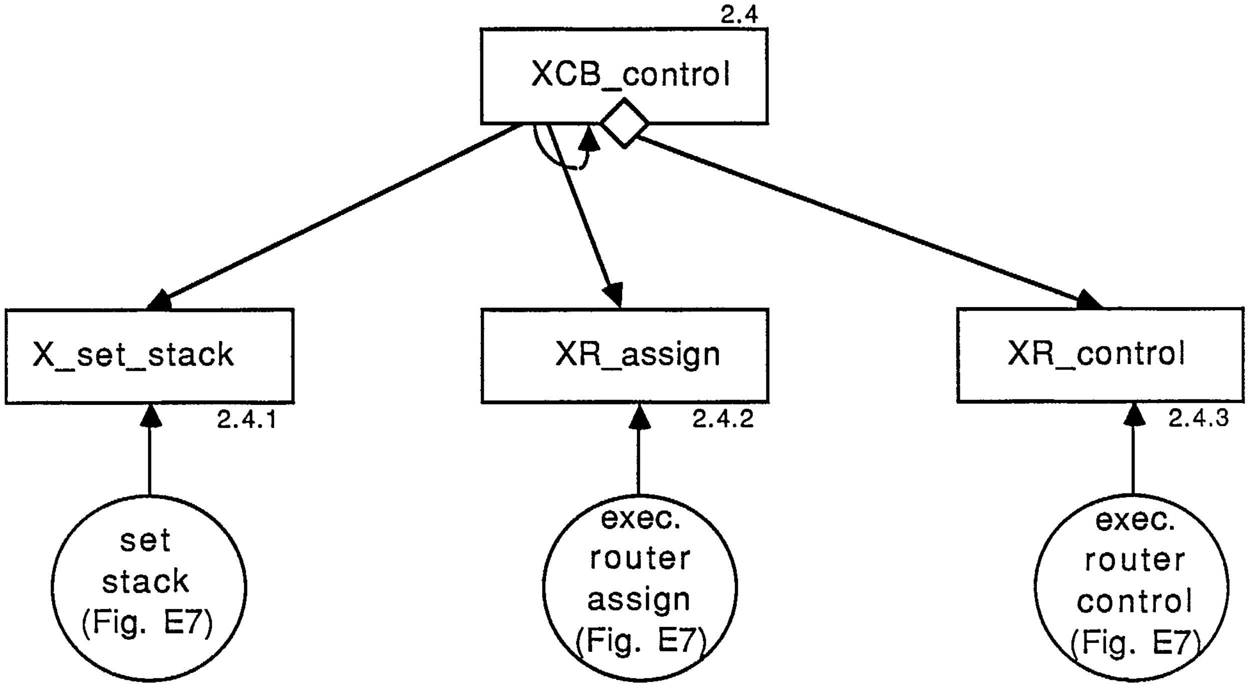


Figure E7: XCB_control

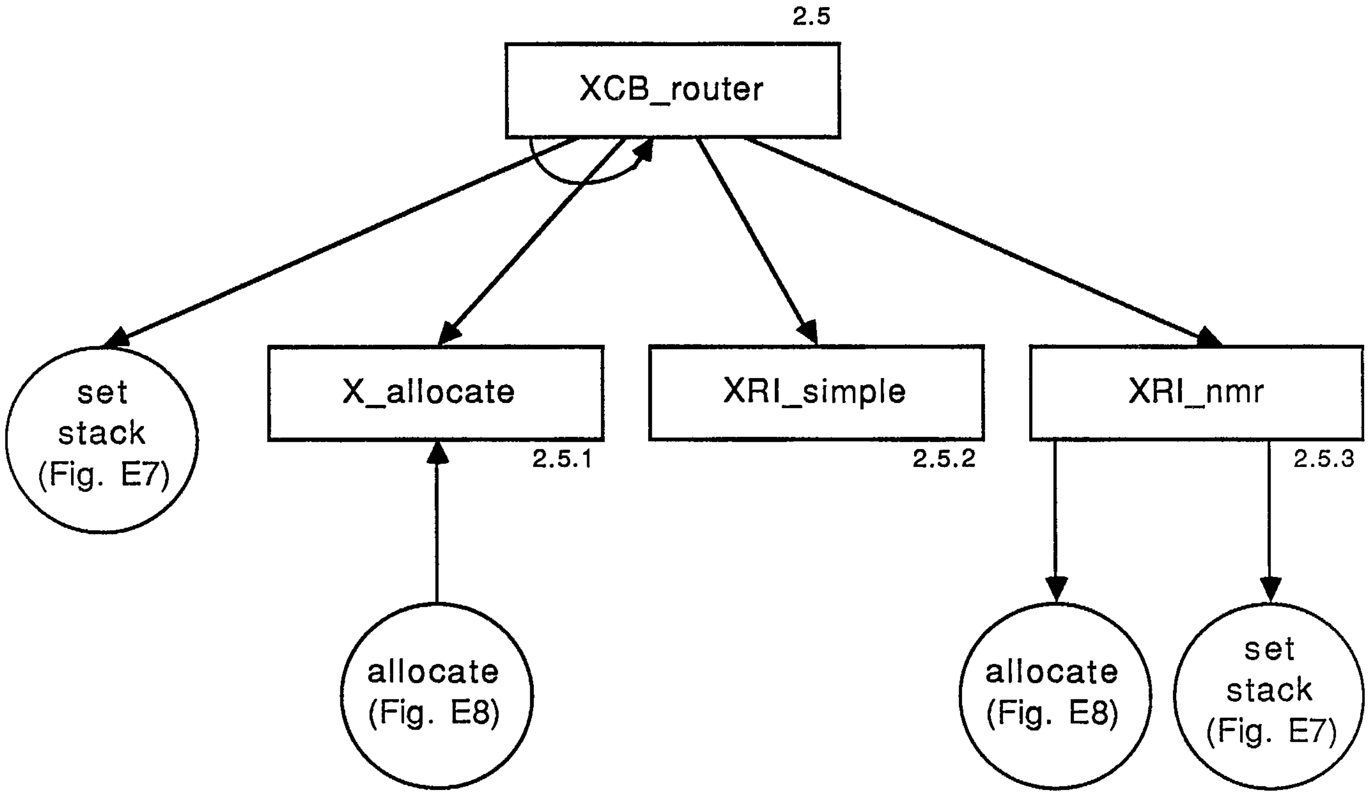


Figure E8: XCB_router

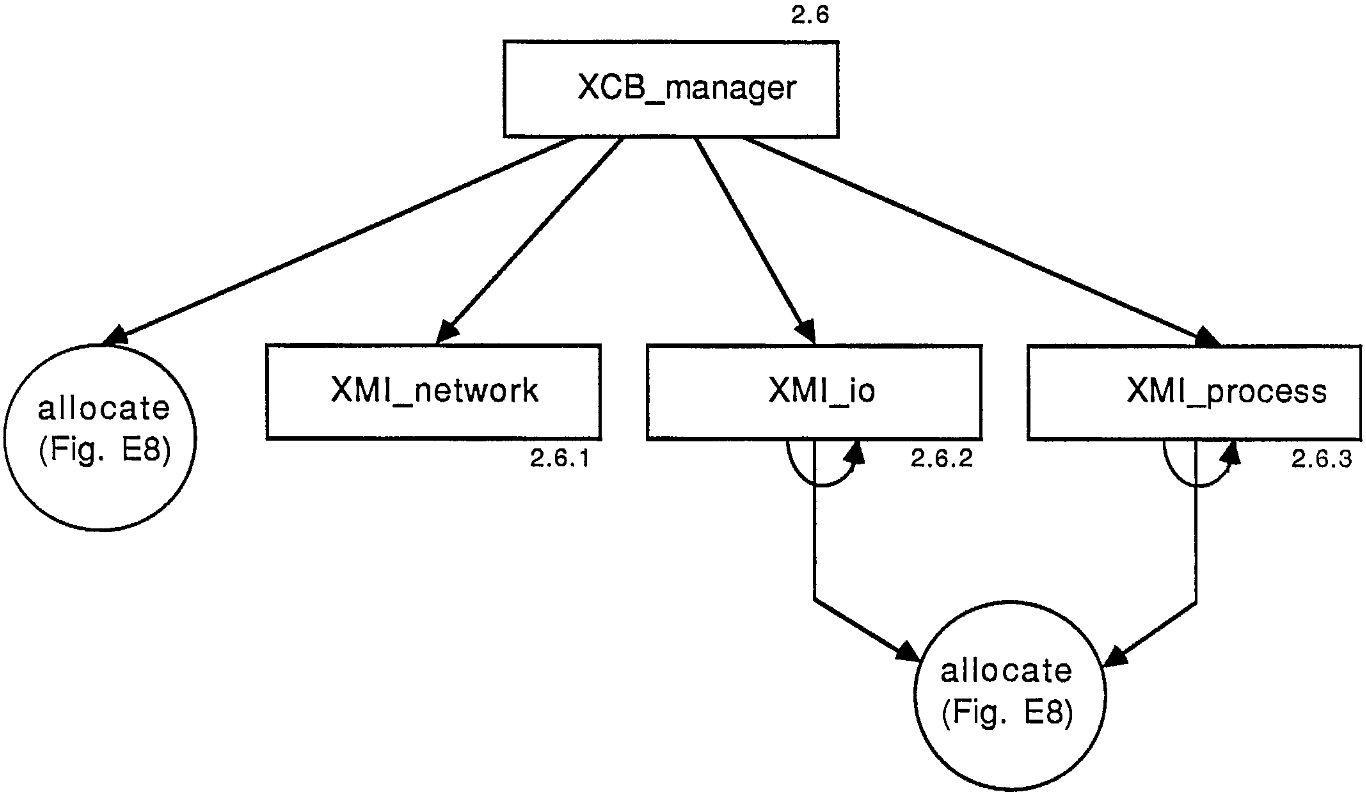


Figure E9: XCB_manager

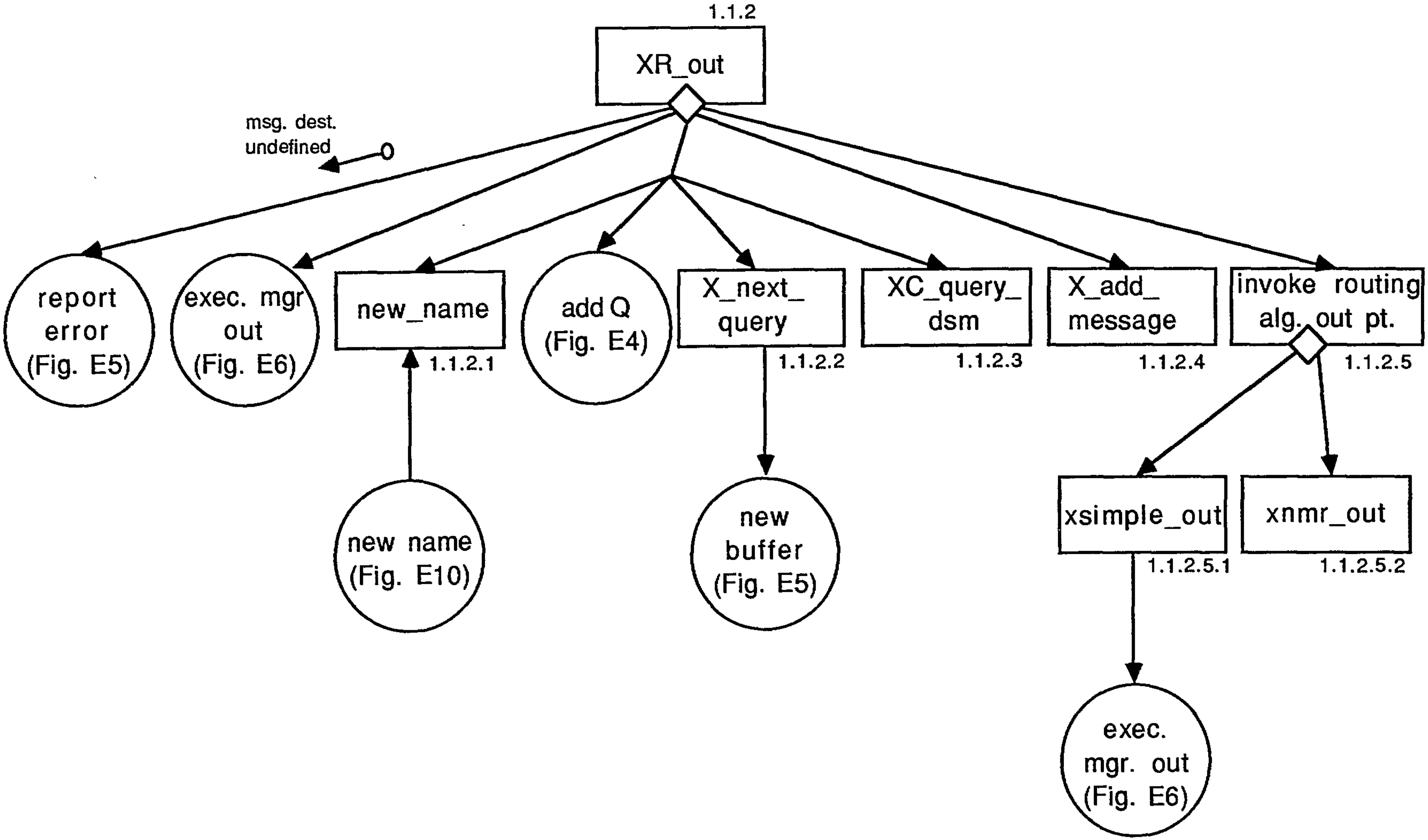


Figure E10: XR_out

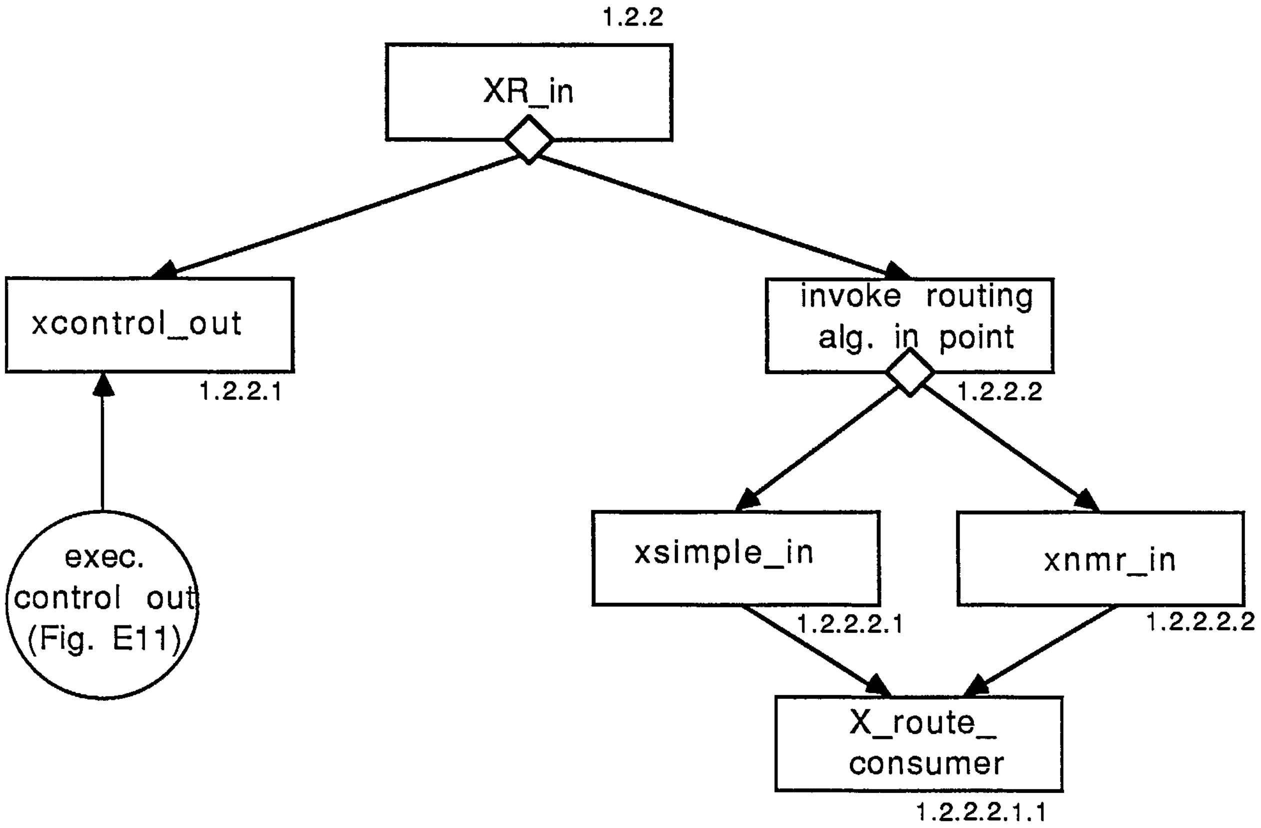


Figure E11: XR_in

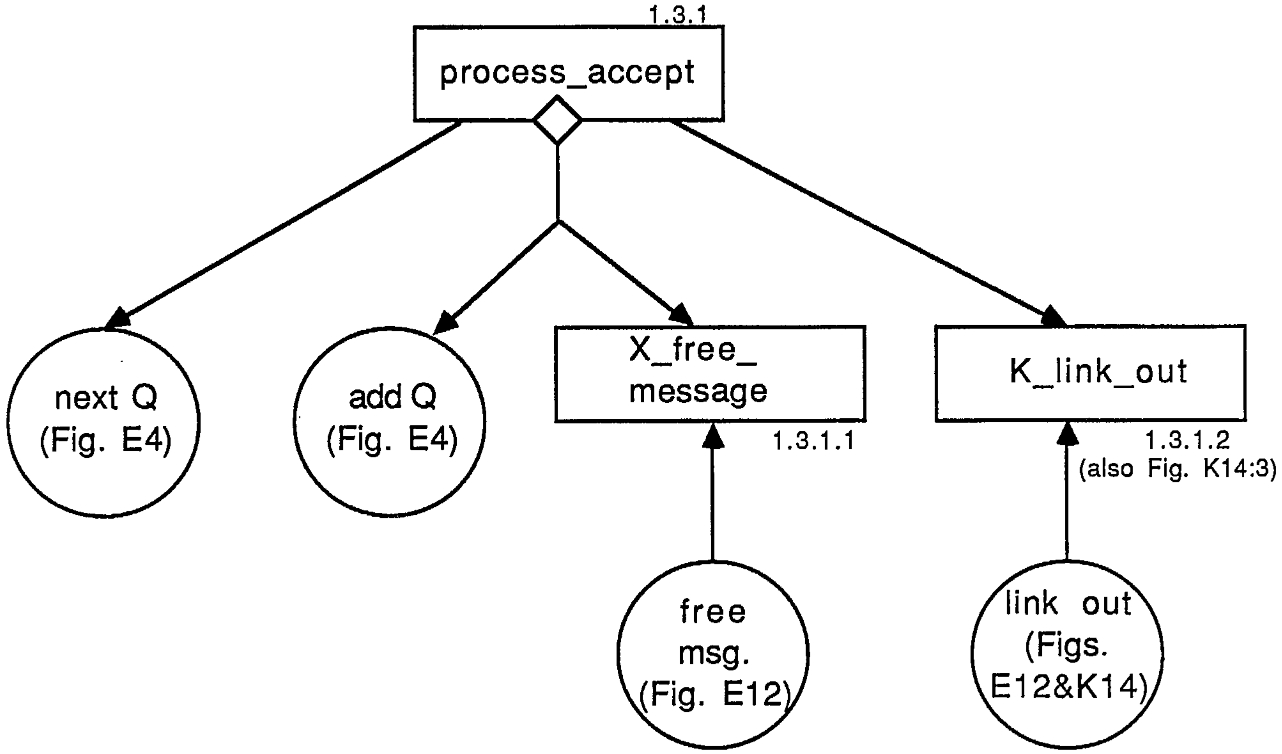


Figure E12: process_accept

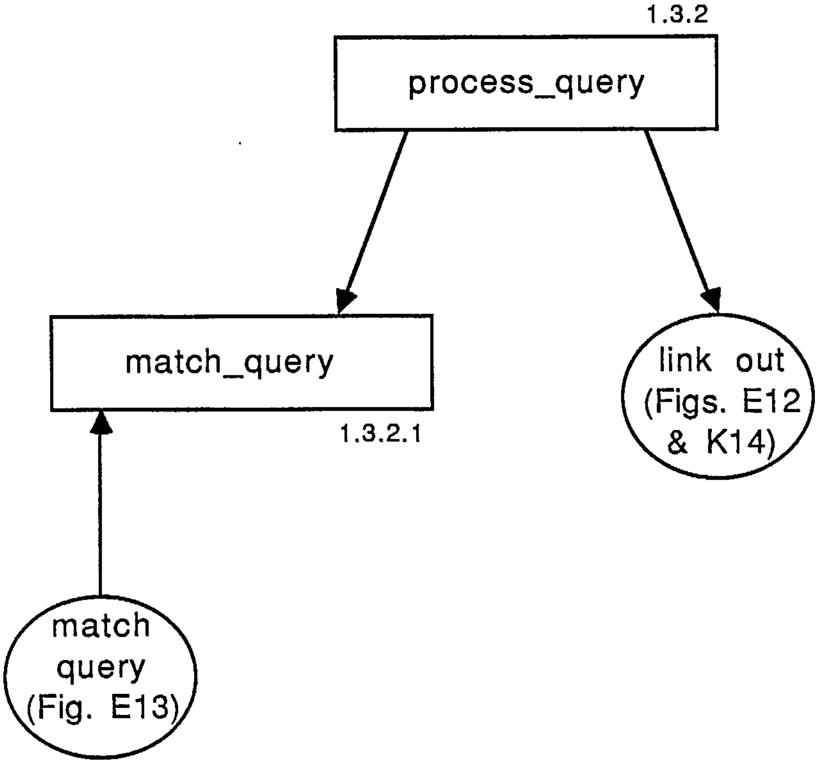


Figure E13: process_query

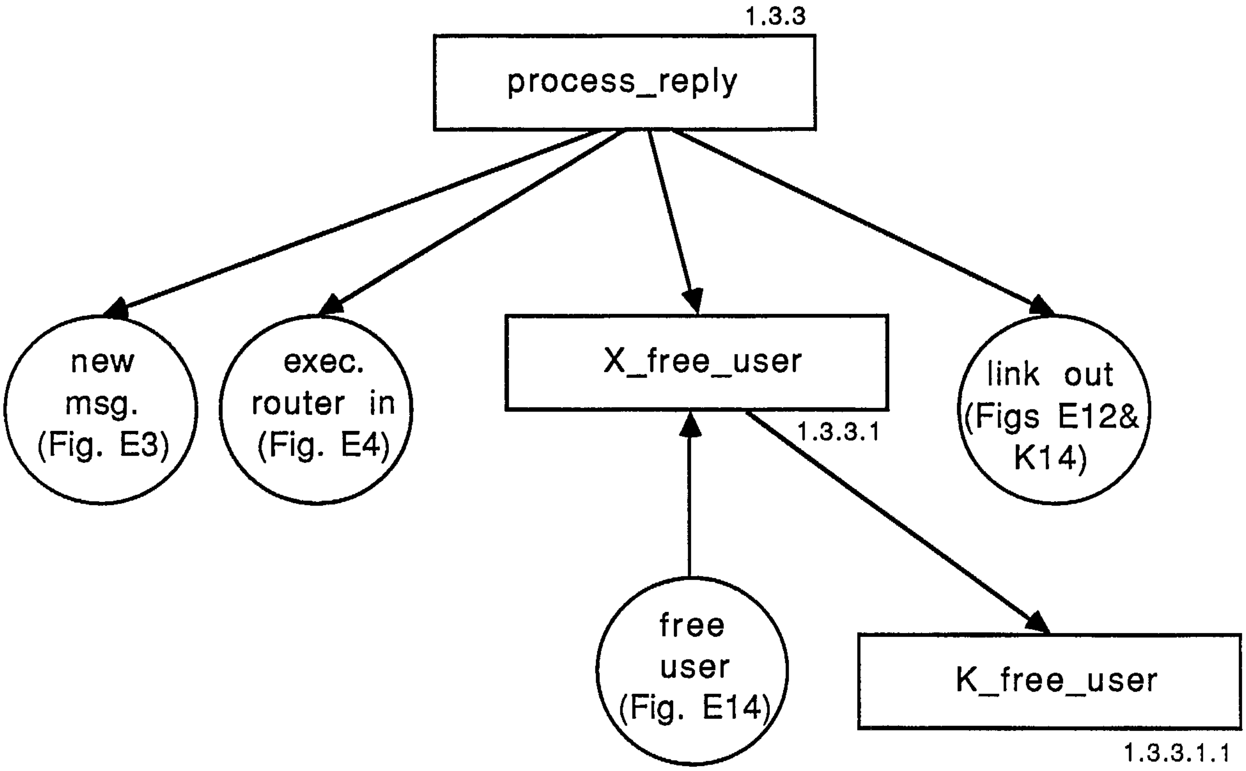


Figure E14: process_reply

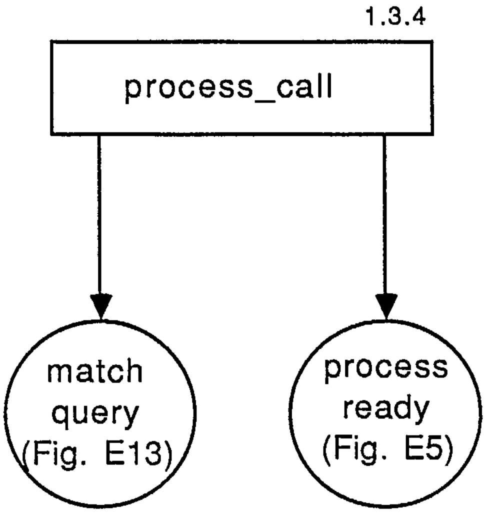


Figure E15: process_call

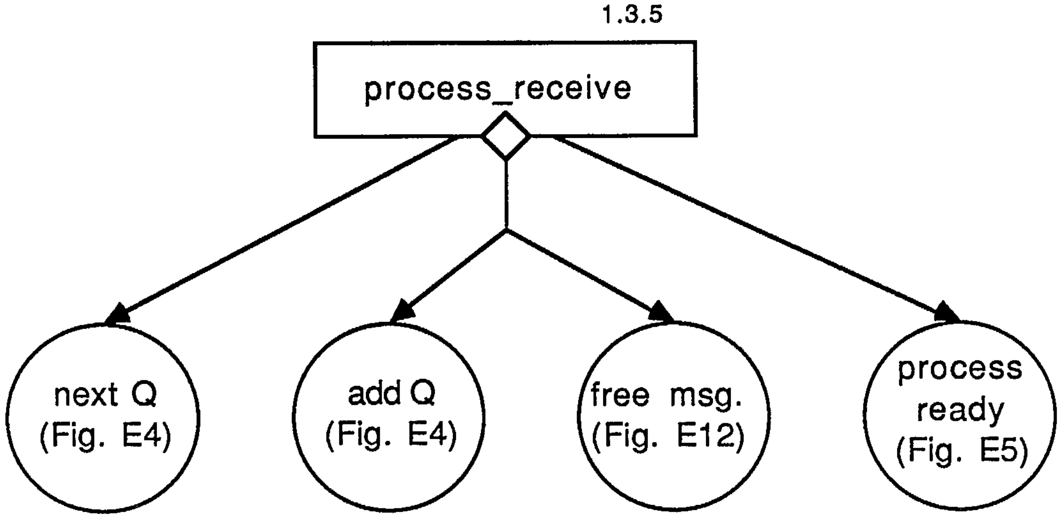


Figure E16: process_receive

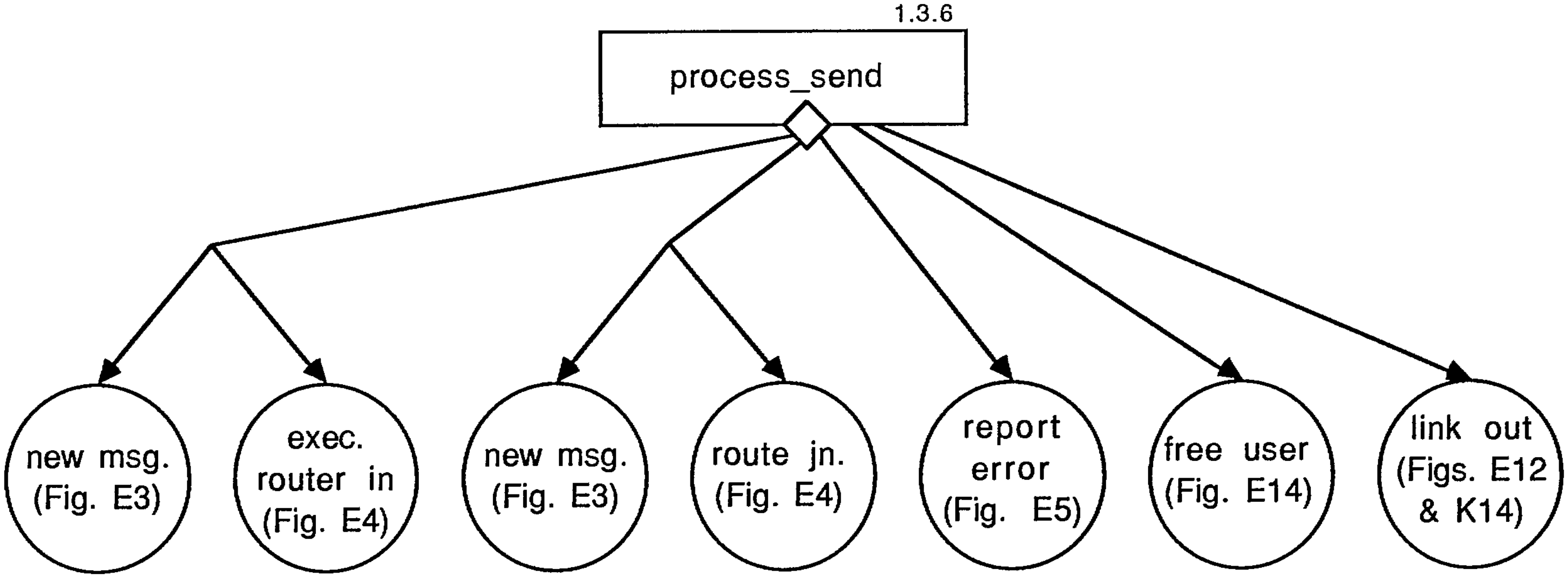


Figure E17: process_send

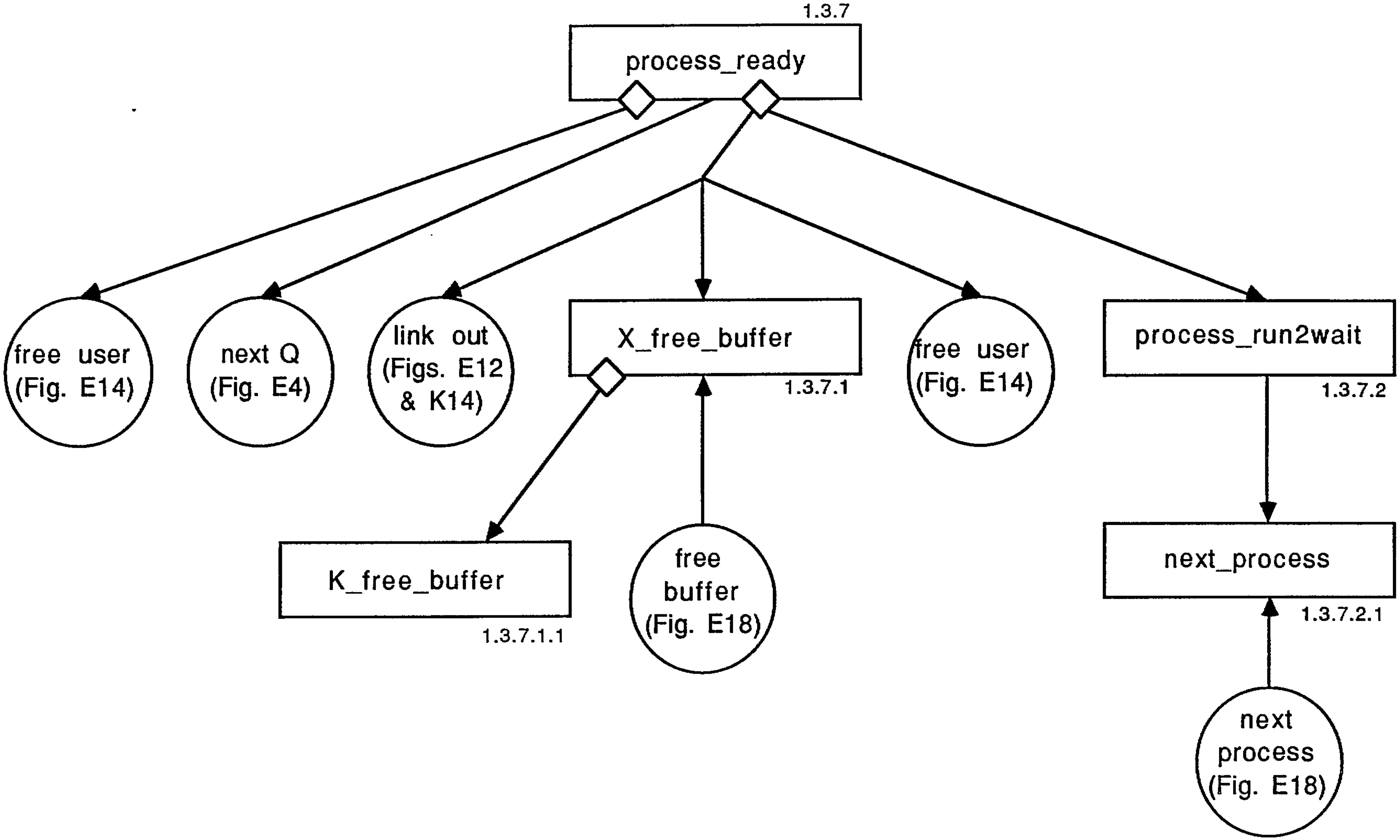


Figure E18: process_ready

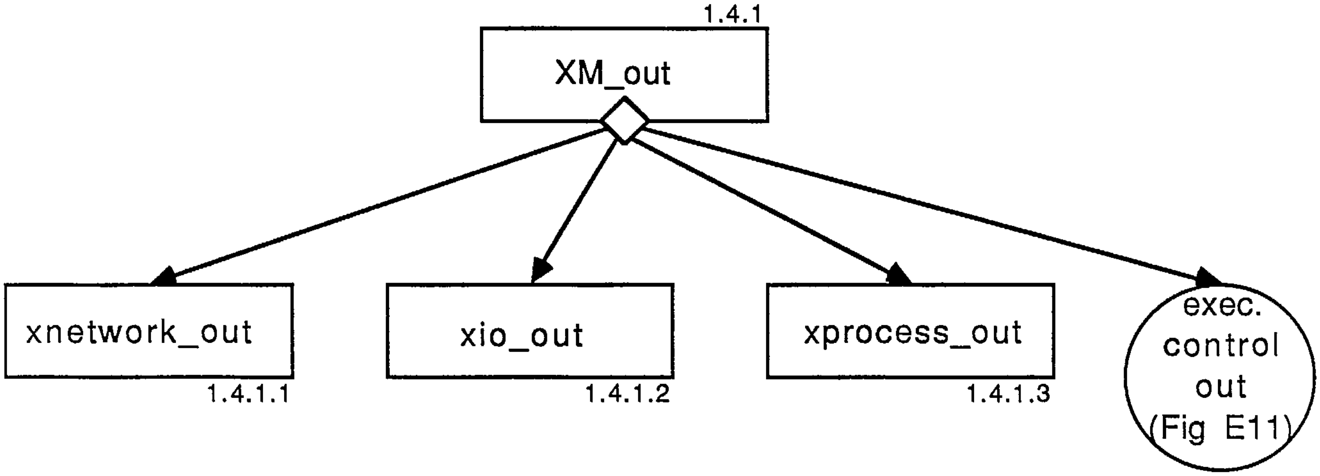


Figure E19: XM_out

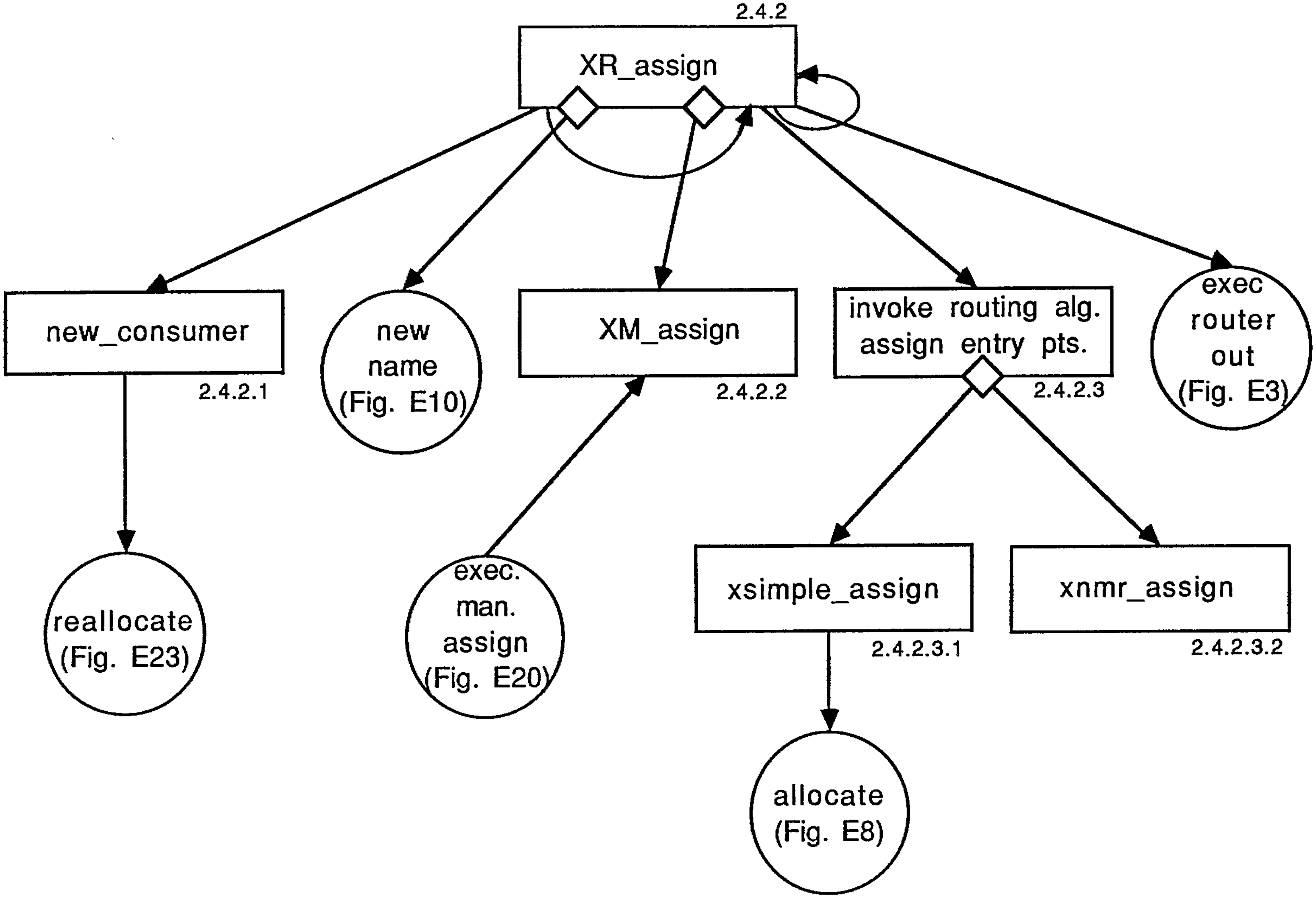


Figure E20: XR_assign

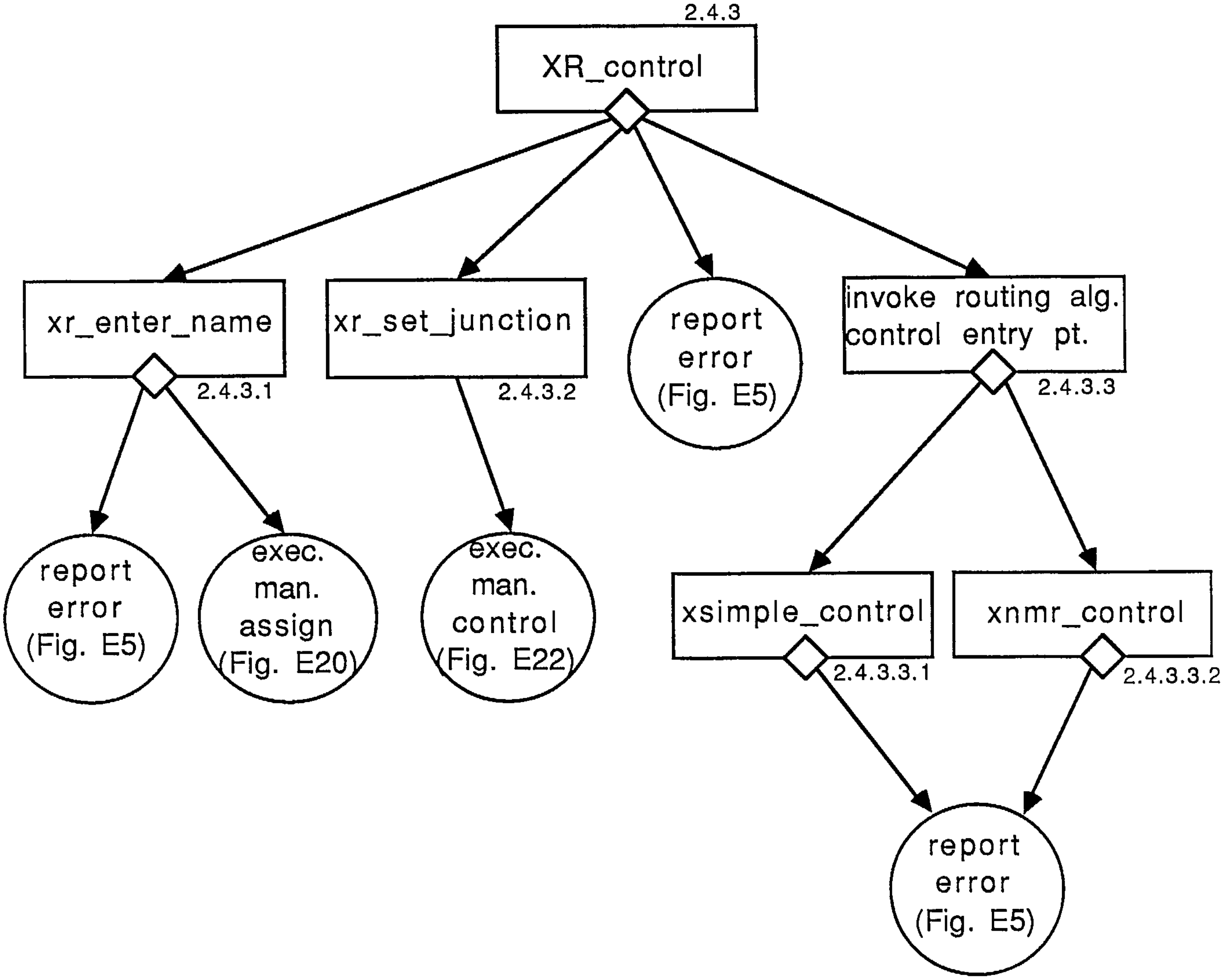


Figure E21: XR_control

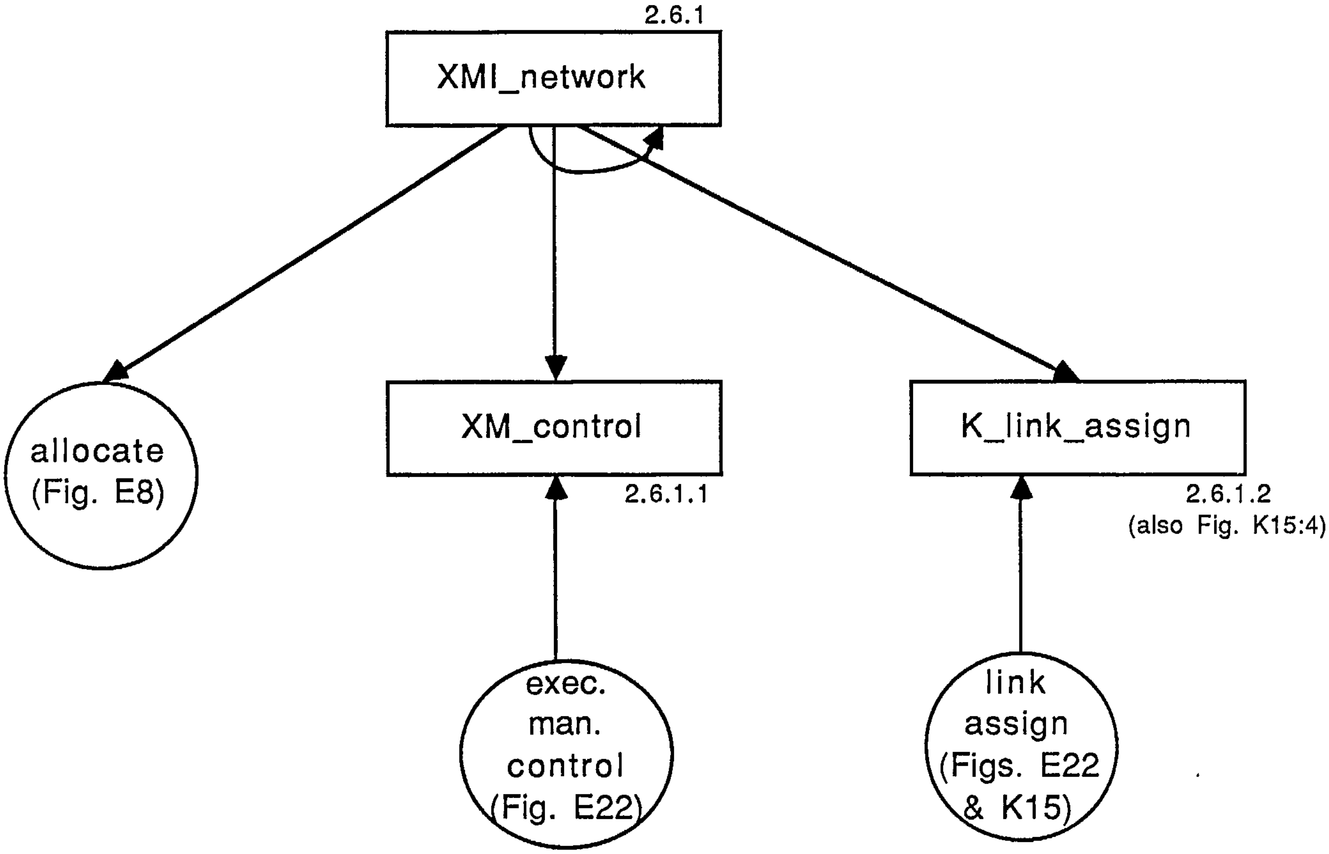


Figure E22: XMI_network

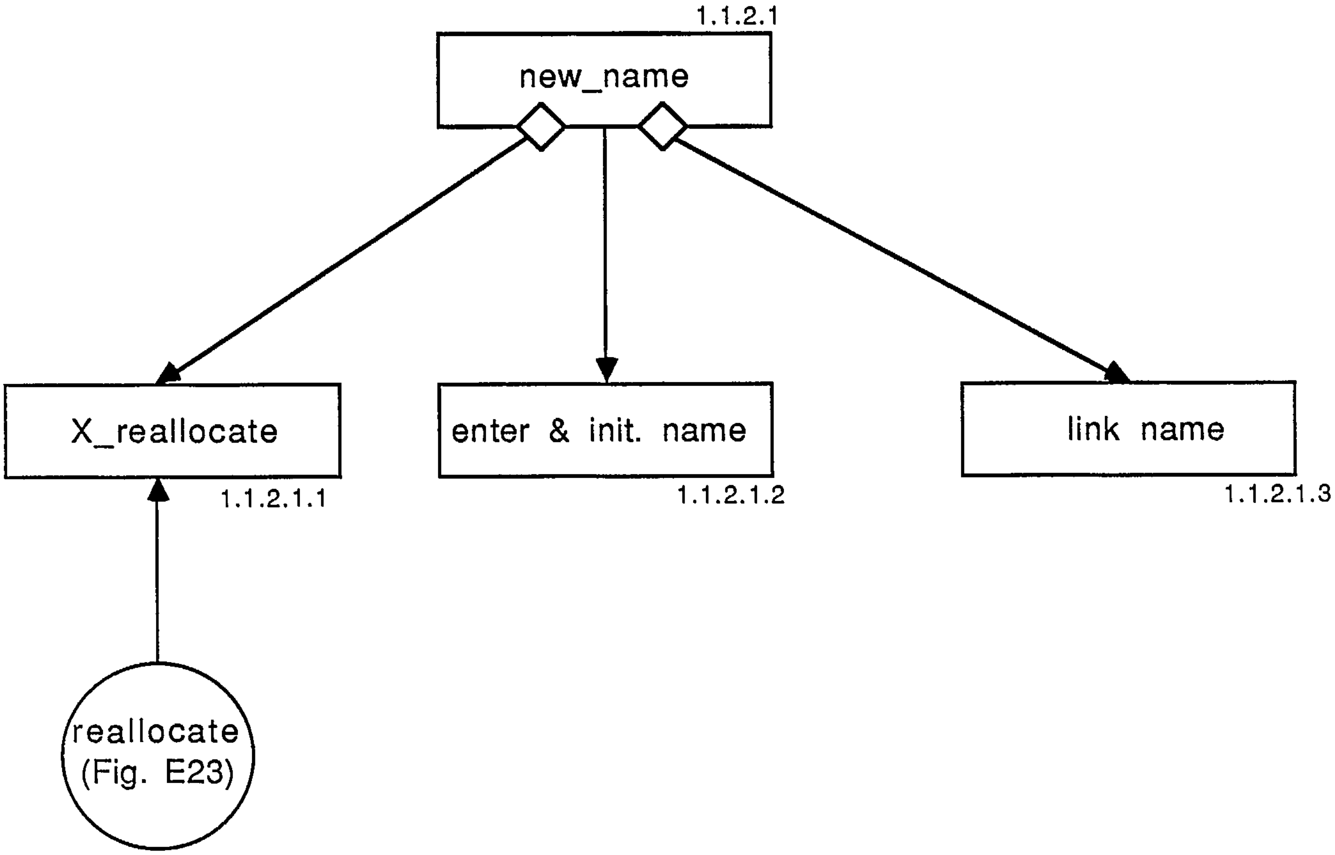


Figure E23: new_name

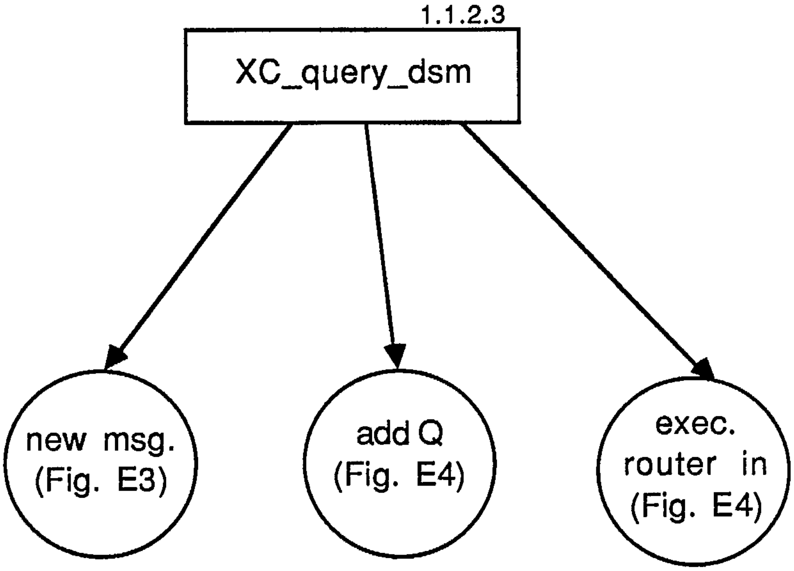


Figure E24: XC_query_dsm

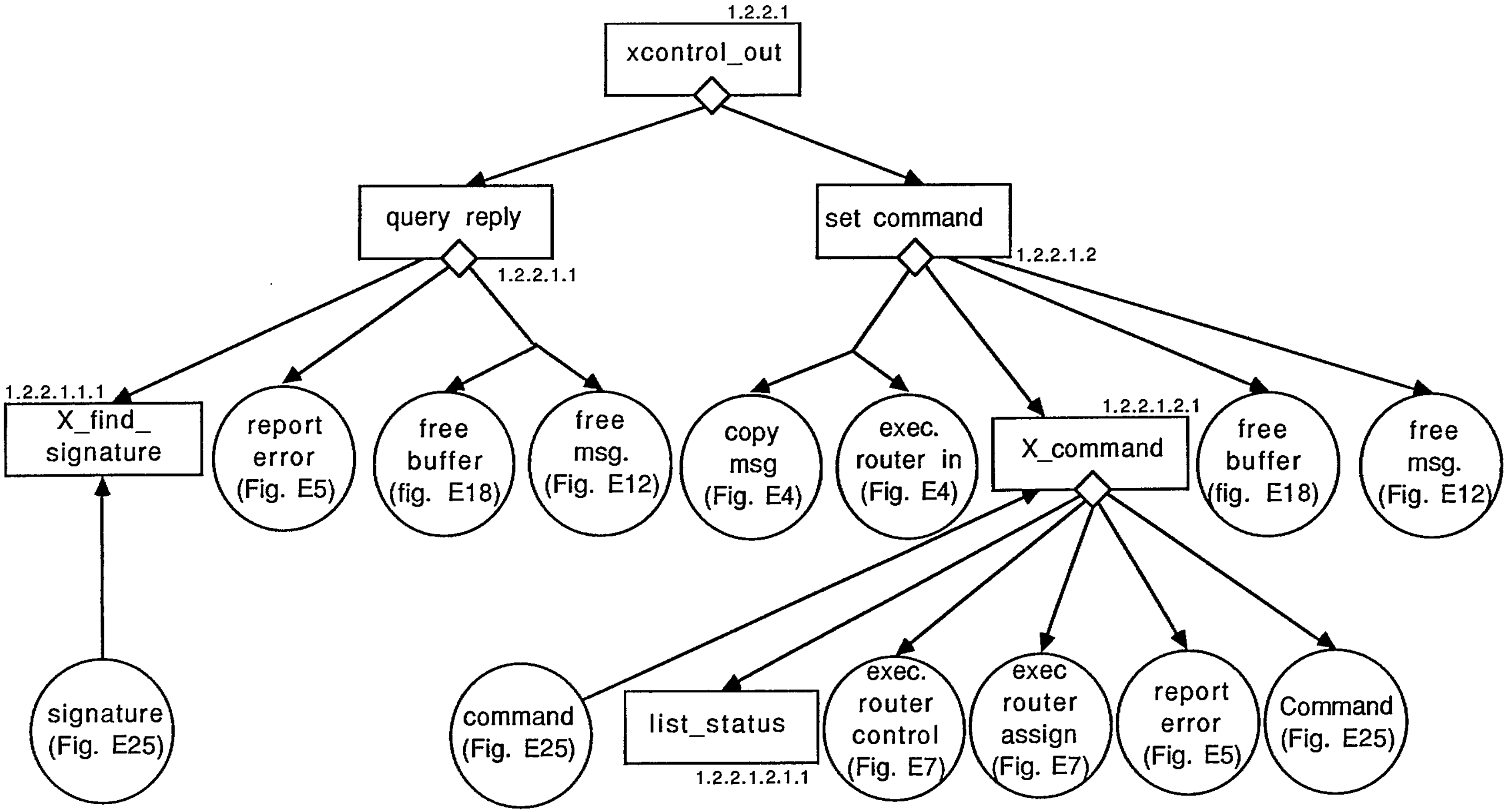


Figure E25: xcontrol_out

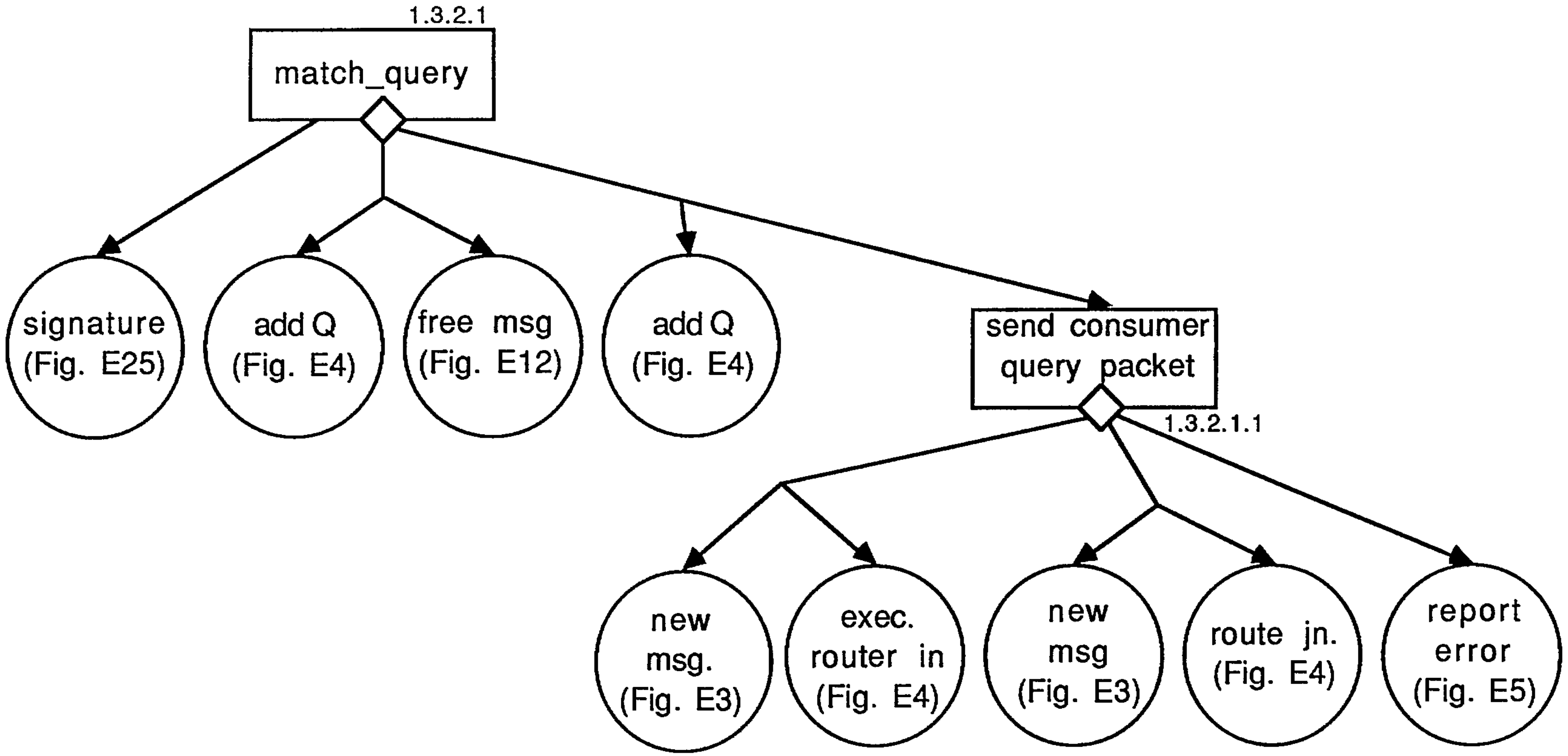


Figure E26: match_query

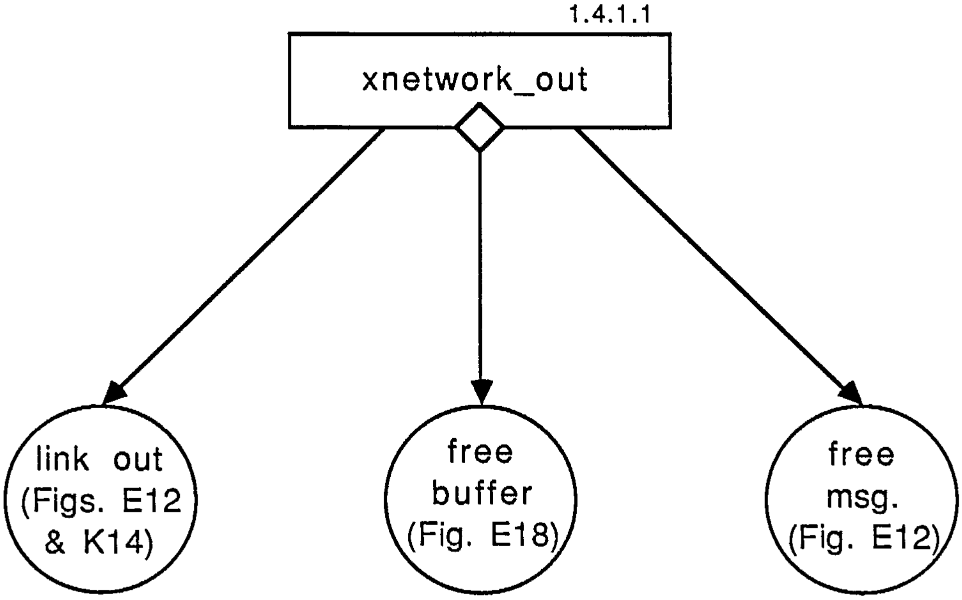


Figure E27: xnetwork_out

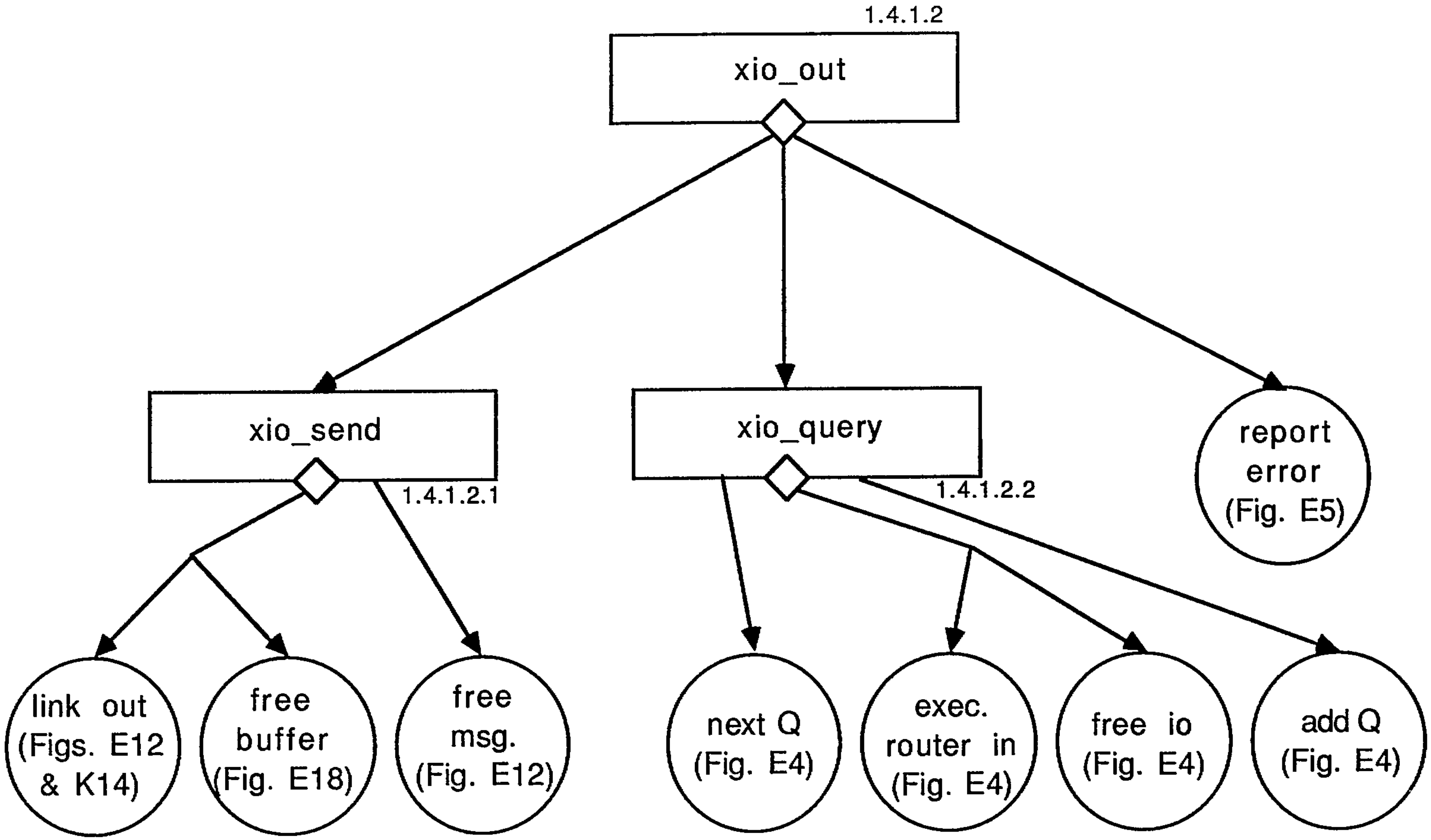


Figure E28: xio_out

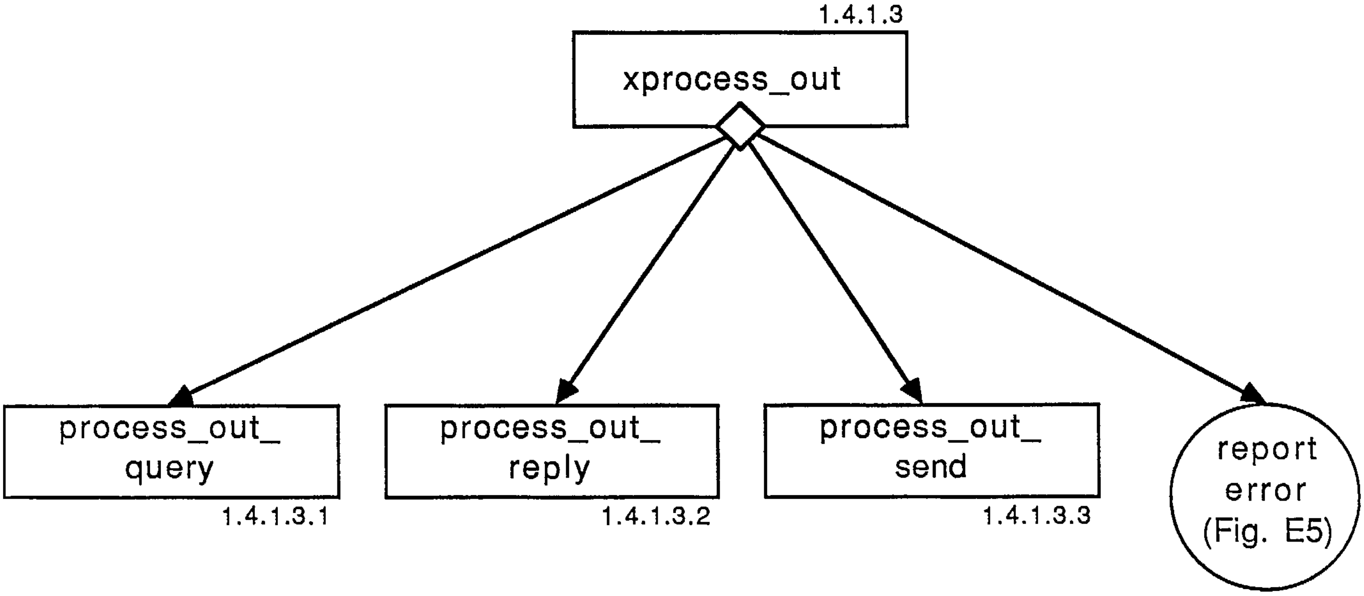


Figure E29: xprocess_out

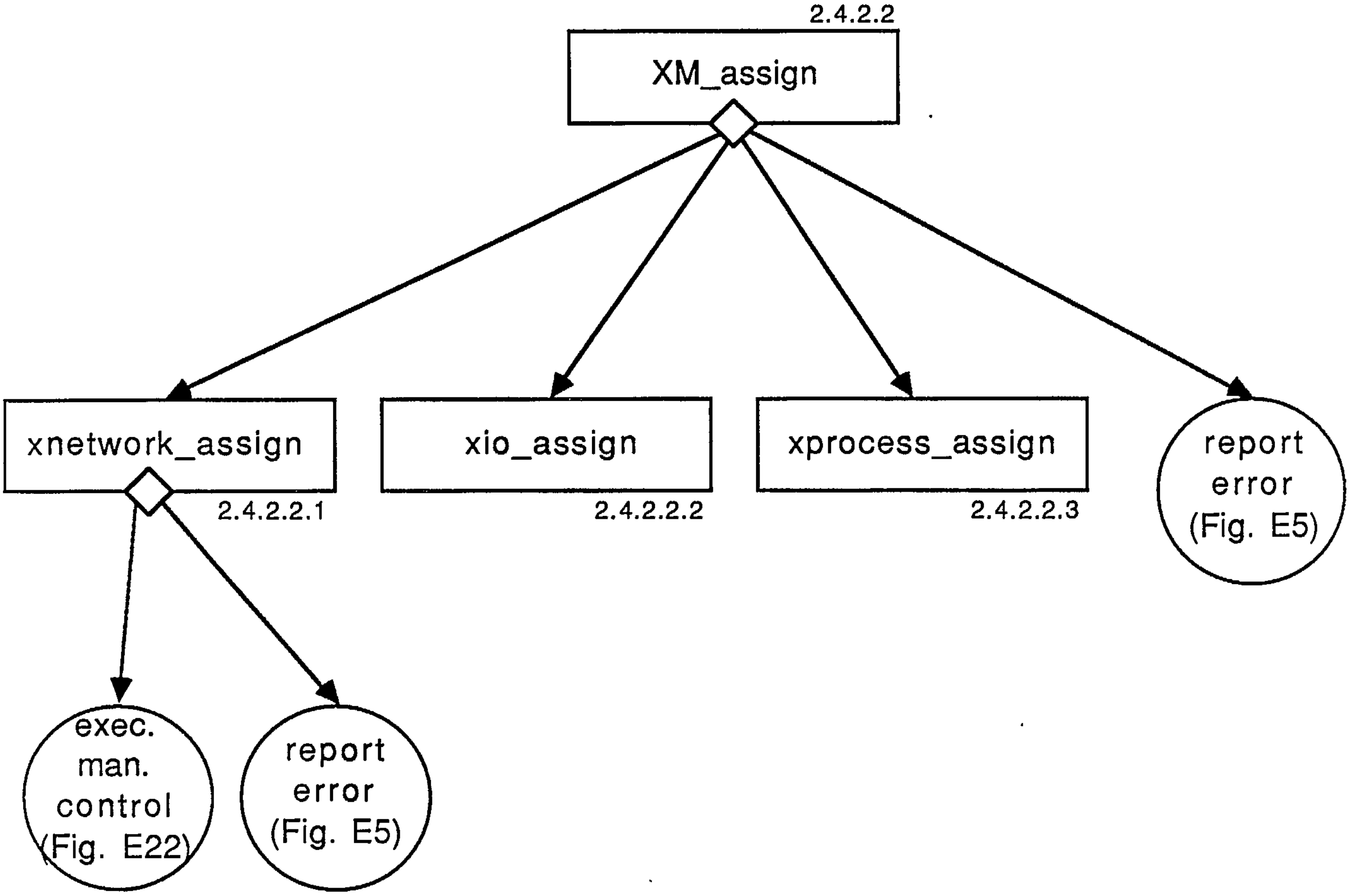


Figure E30: XM_assign

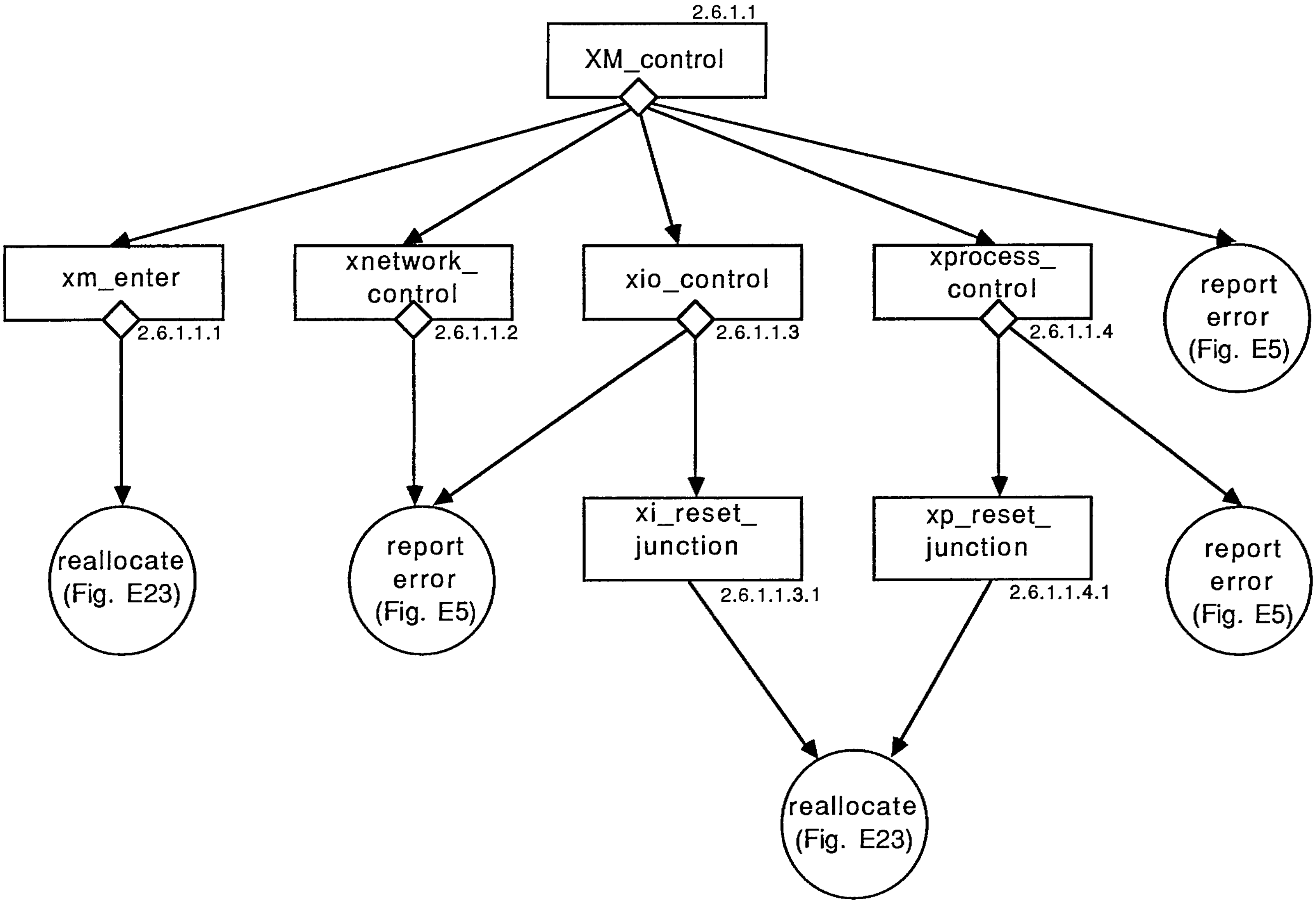


Figure E31: XM_control

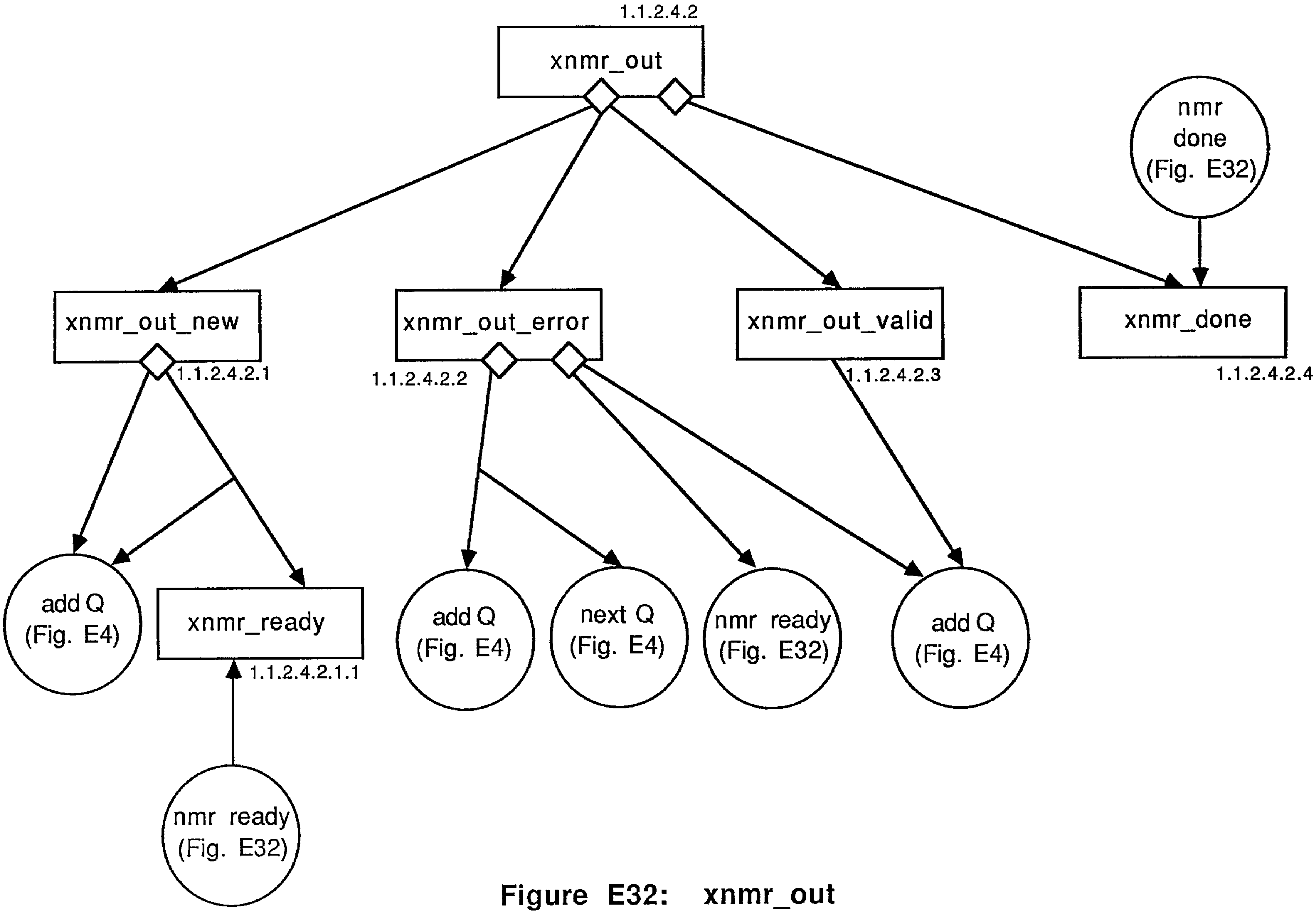


Figure E32: xnmr_out

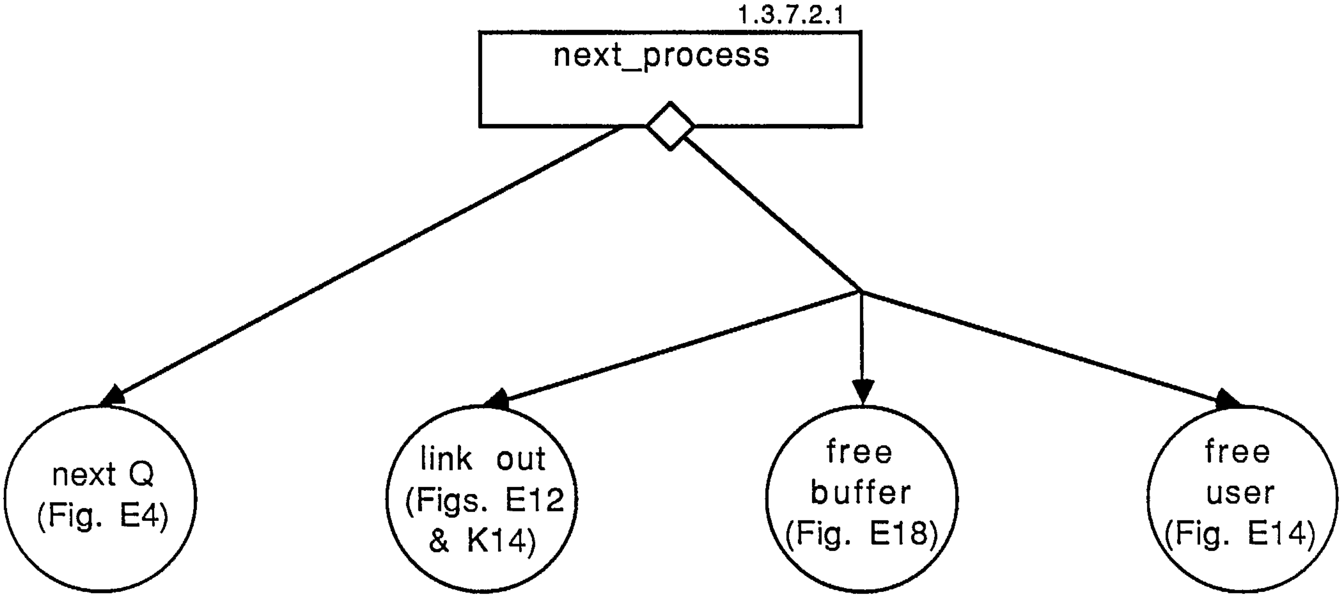


Figure E33: next_process

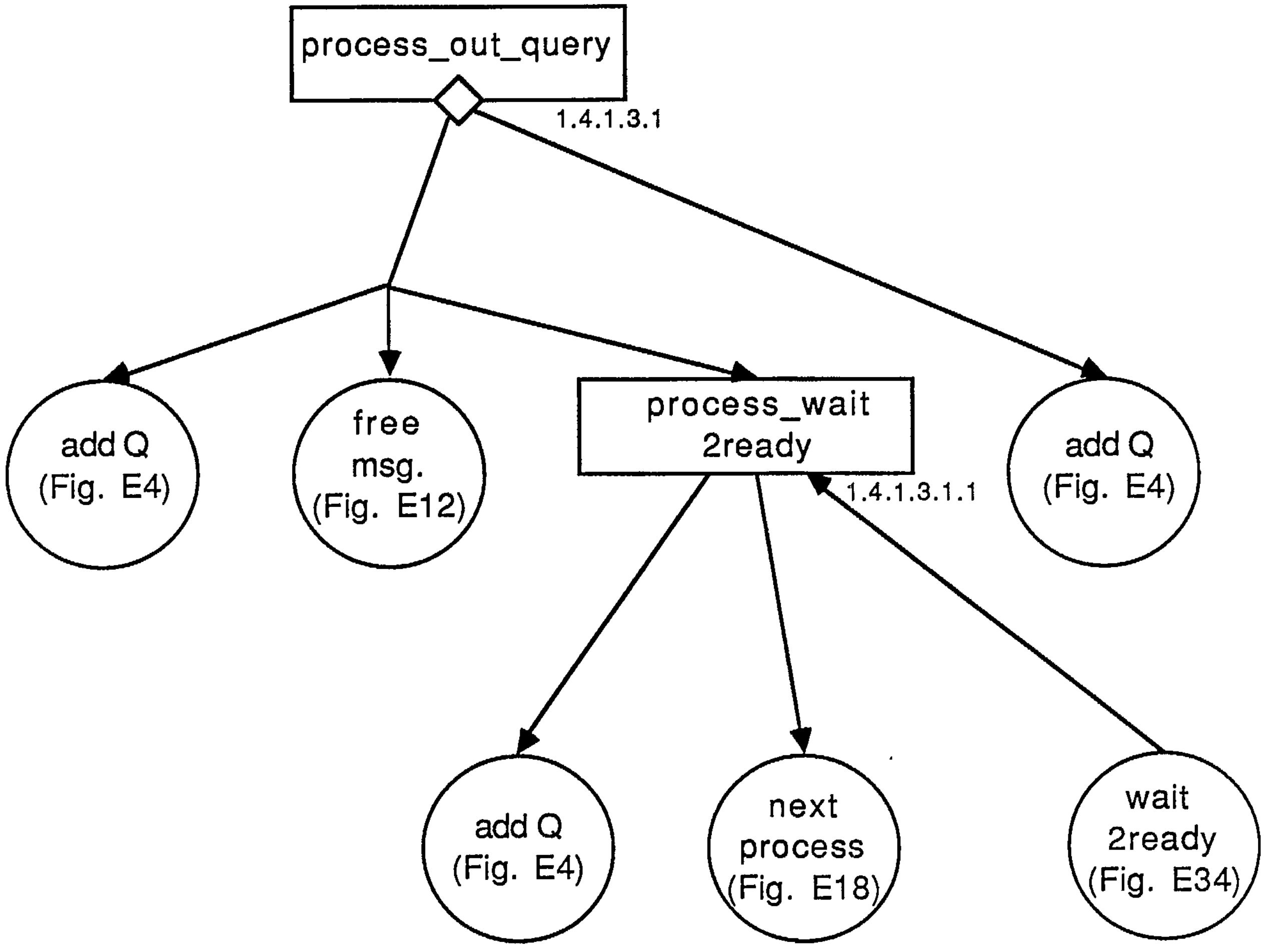


Figure E34: process_out_query

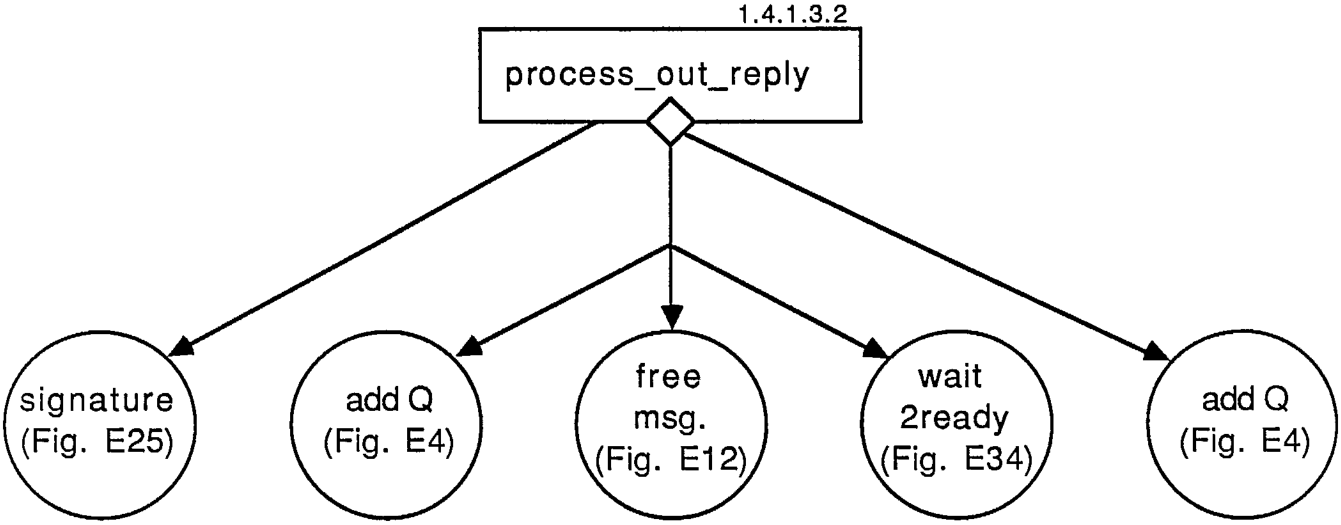


Figure E35: process_out_reply

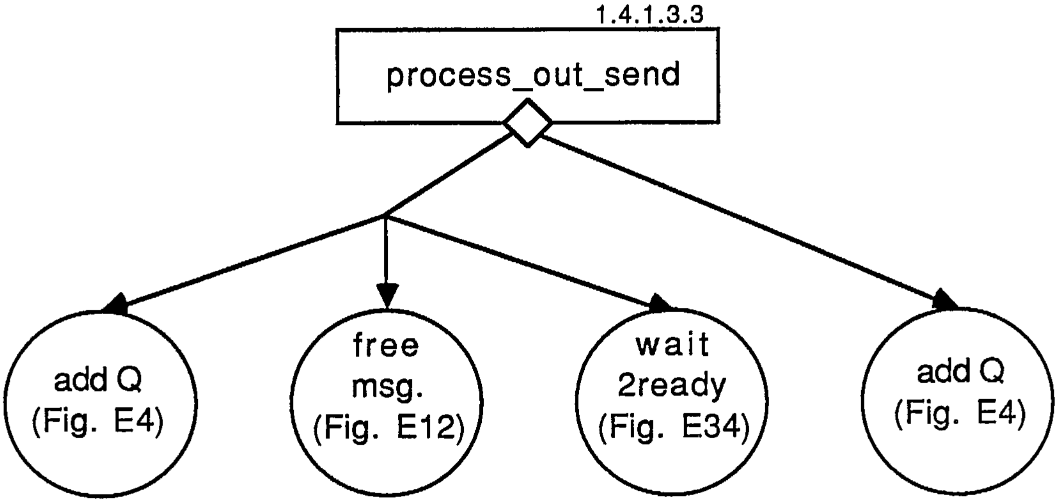


Figure E36: process_out_send

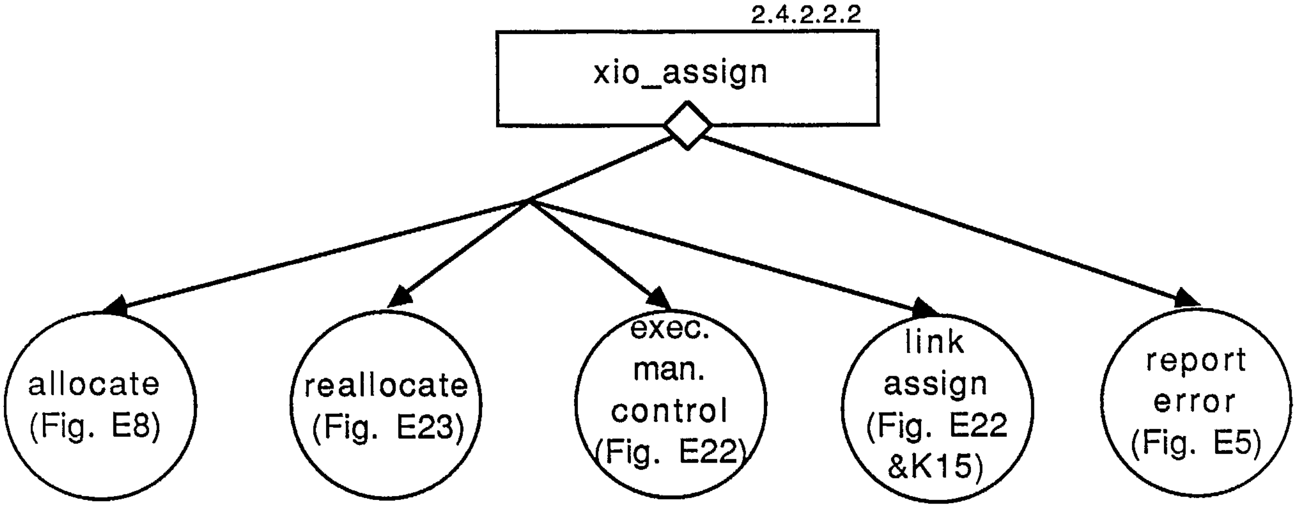


Figure E37: xio_assign

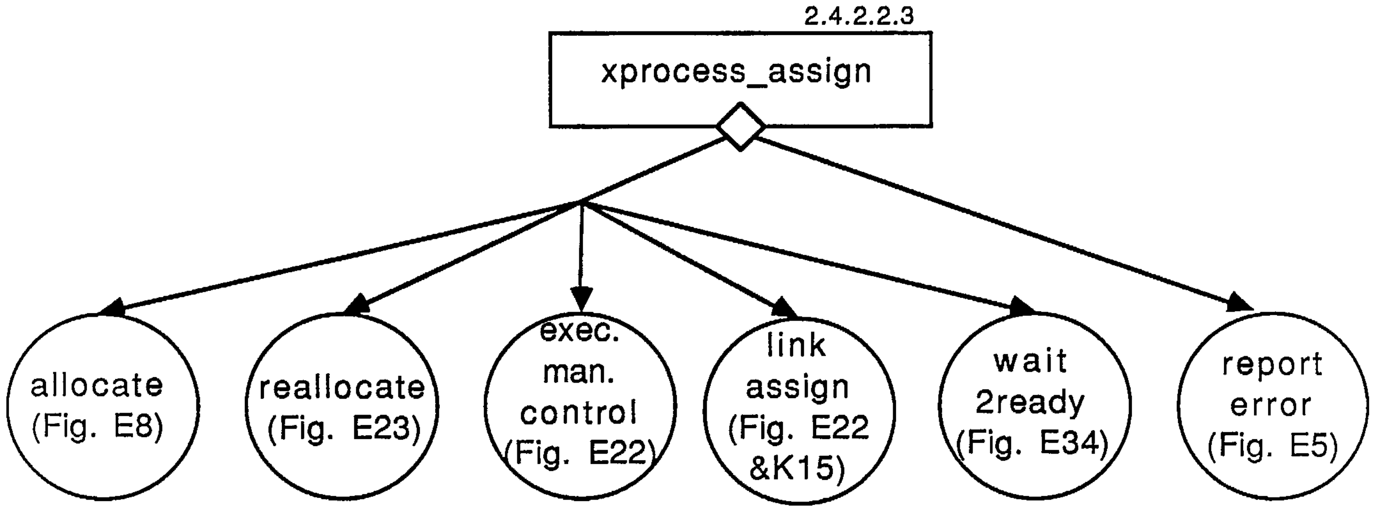


Figure E38: xprocess_assign

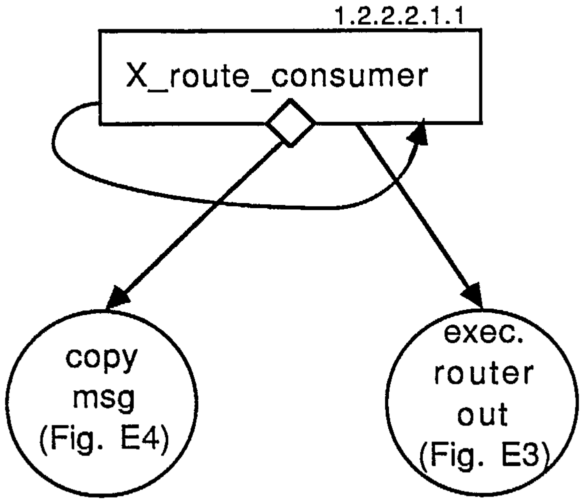


Figure E39: X_route_consumer

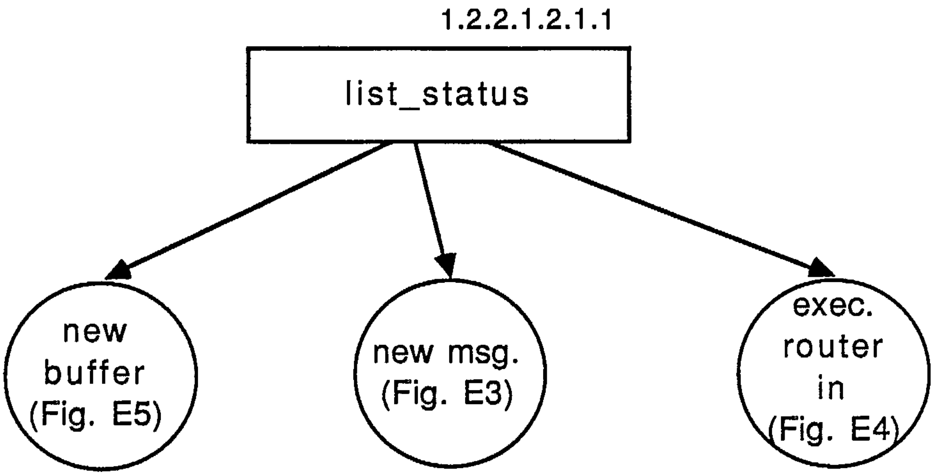


Figure E40: list_status

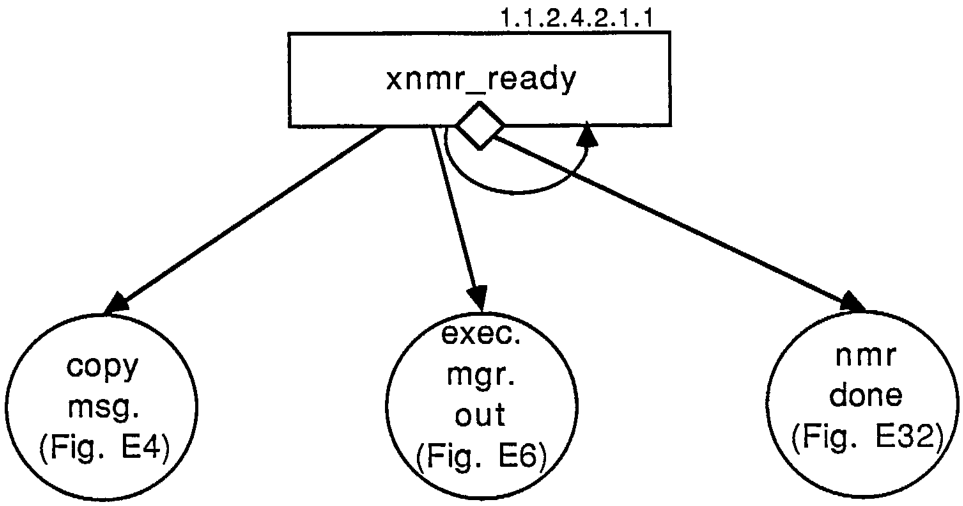


Figure E41: xnmr_ready

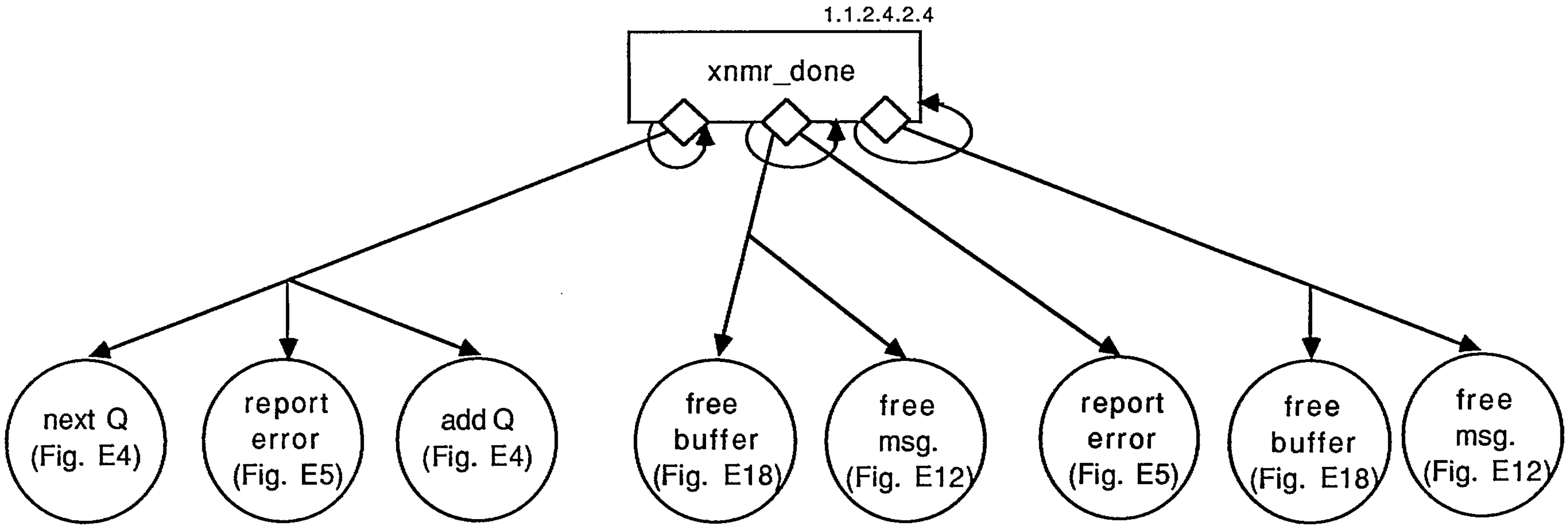


Figure E42: xnmr_done

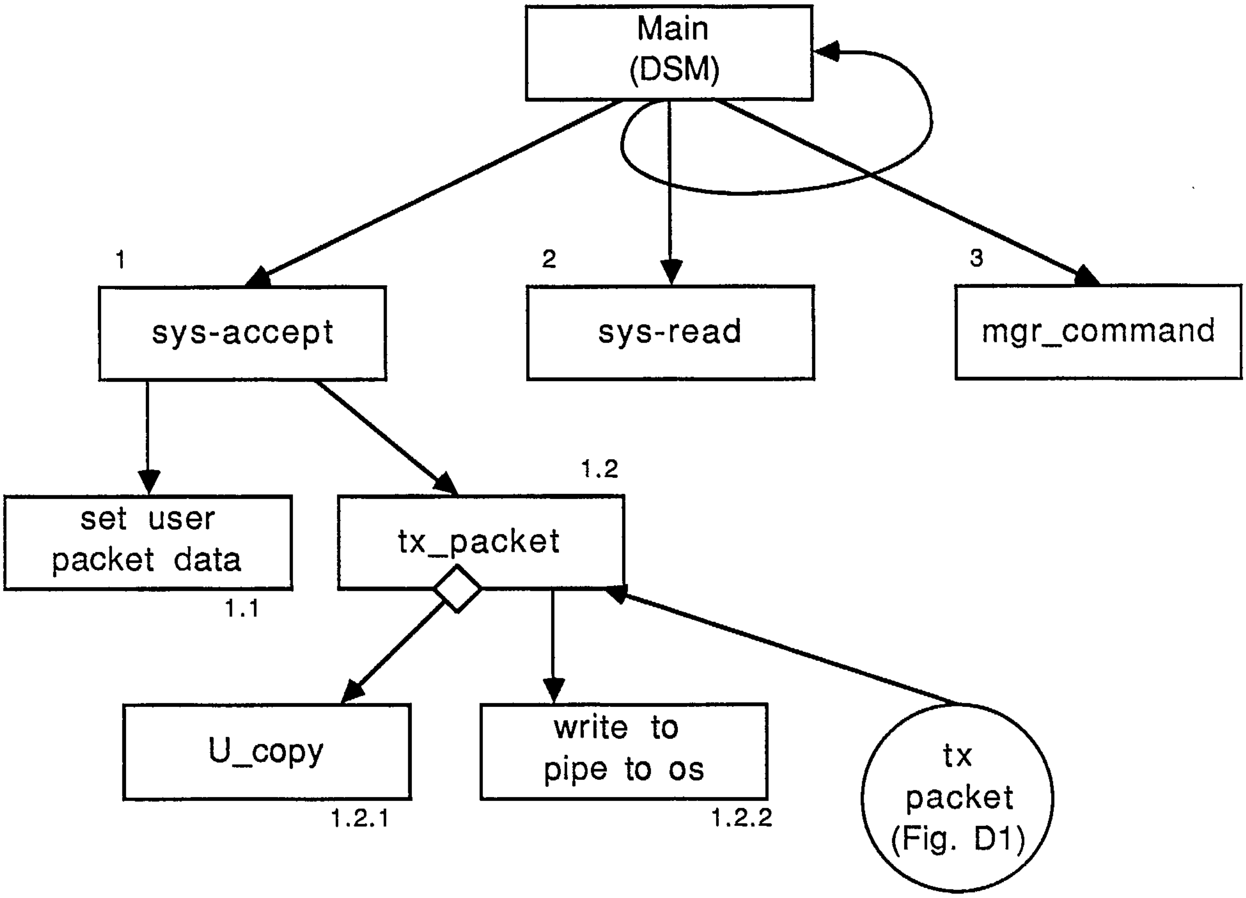


Figure D1: Main (DSM)

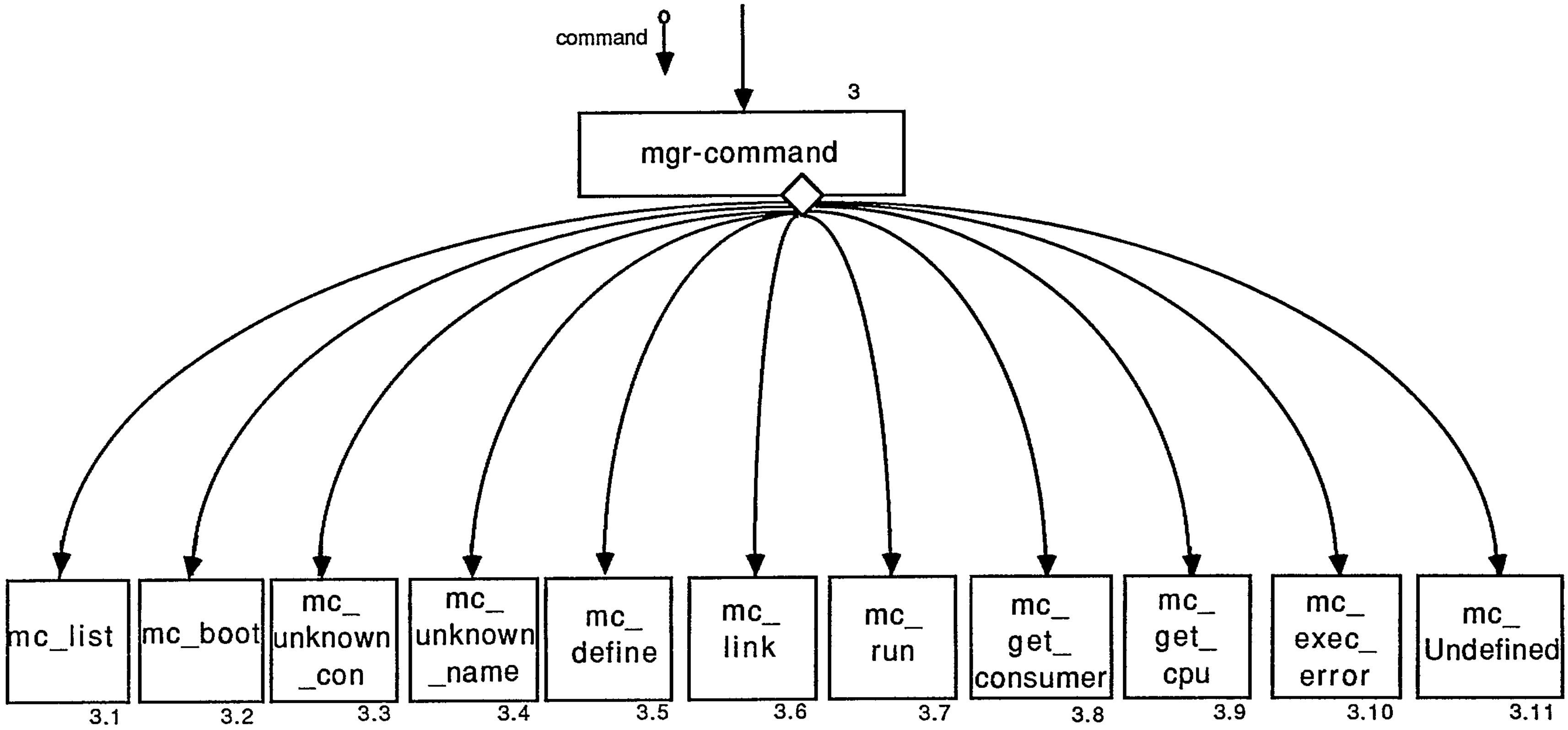


Figure D2: mgr_command

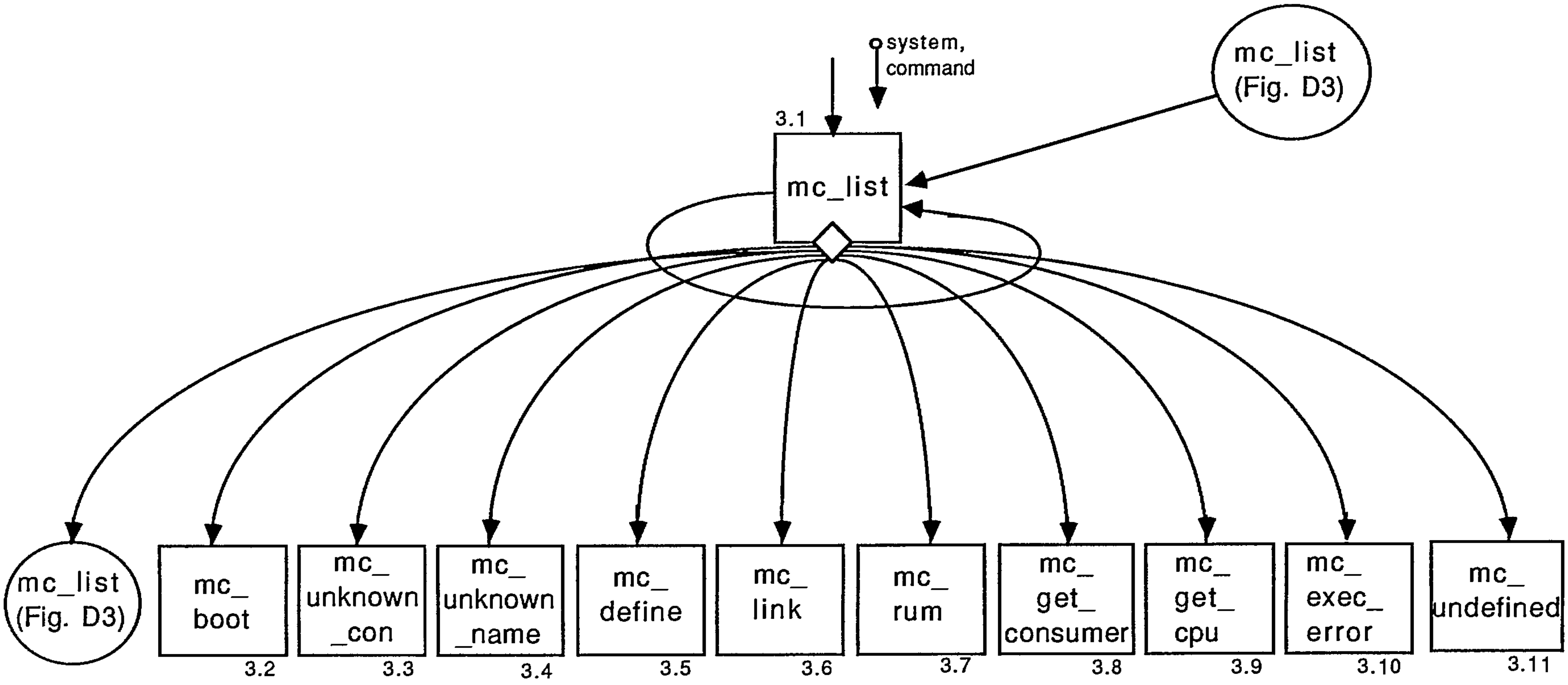


Figure D3: mc_list

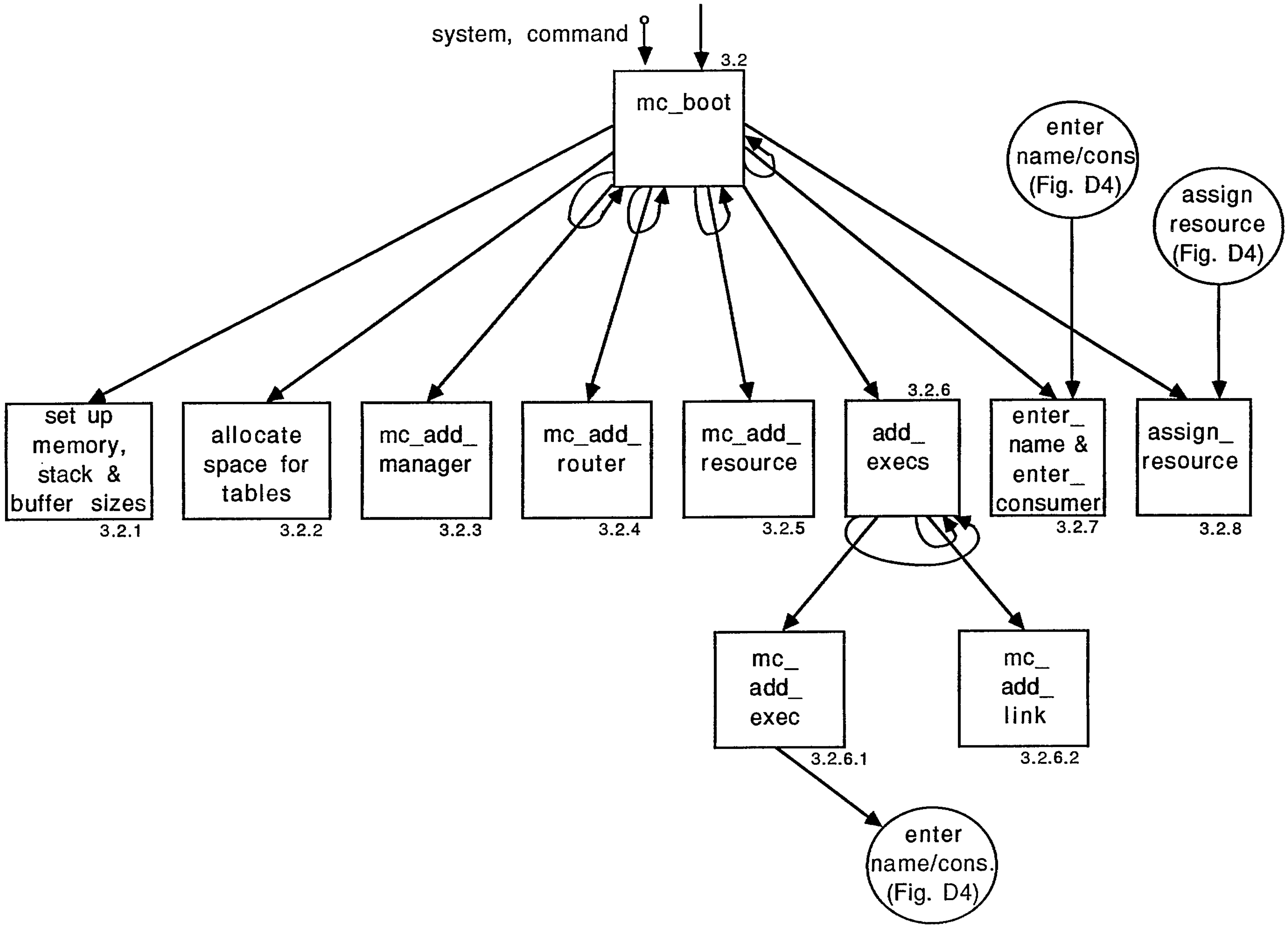


Figure D4: mc_boot

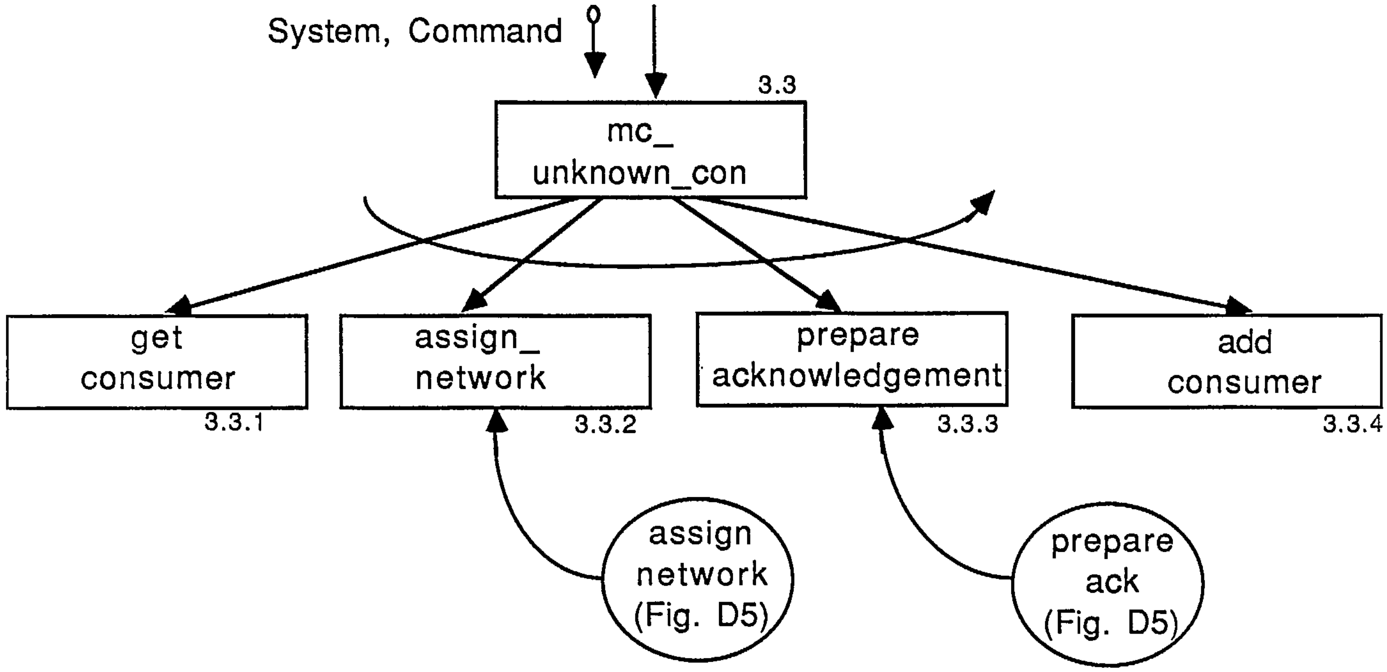


Figure D5: mc_unknown_con

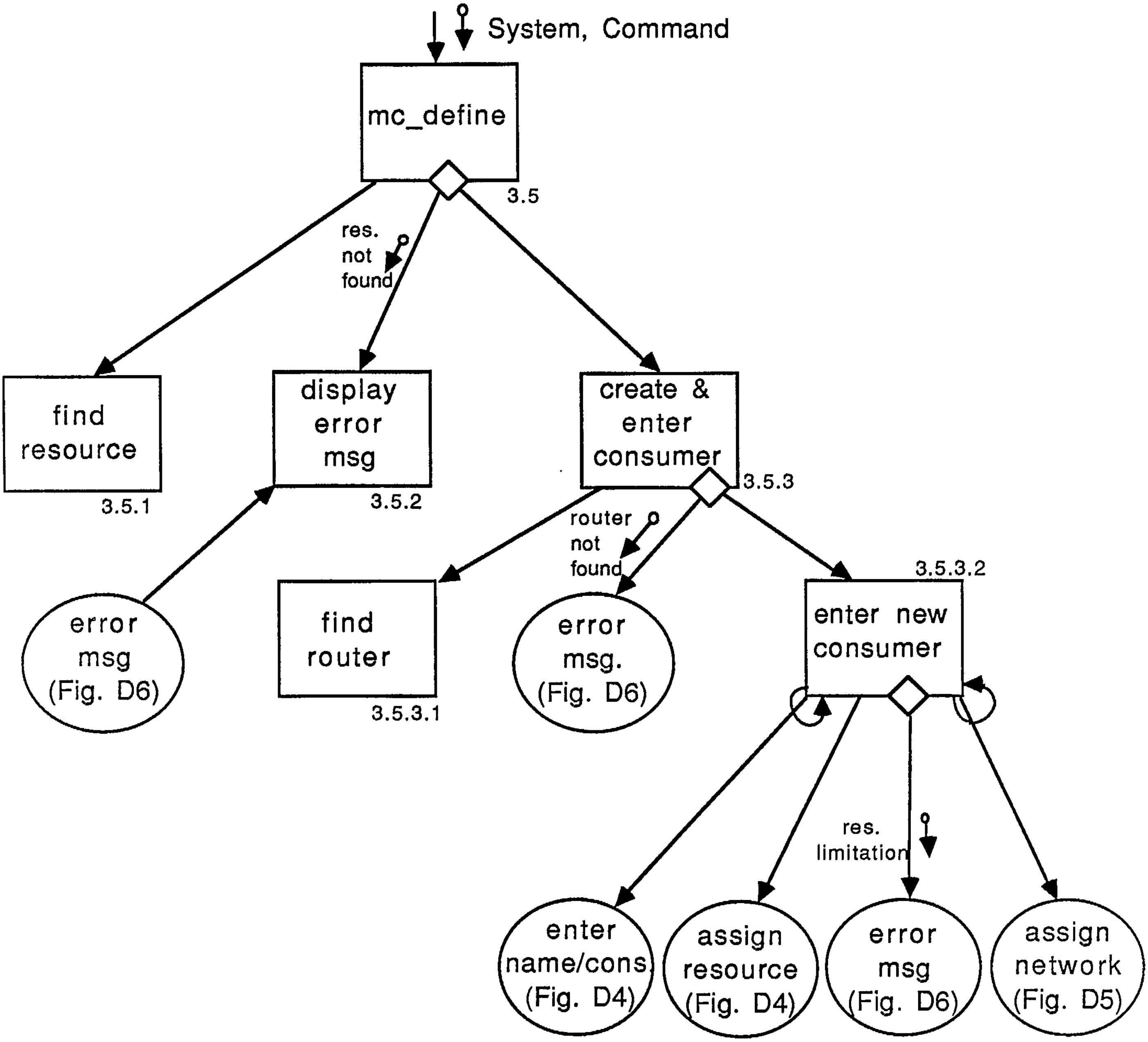


Figure D6: mc_define

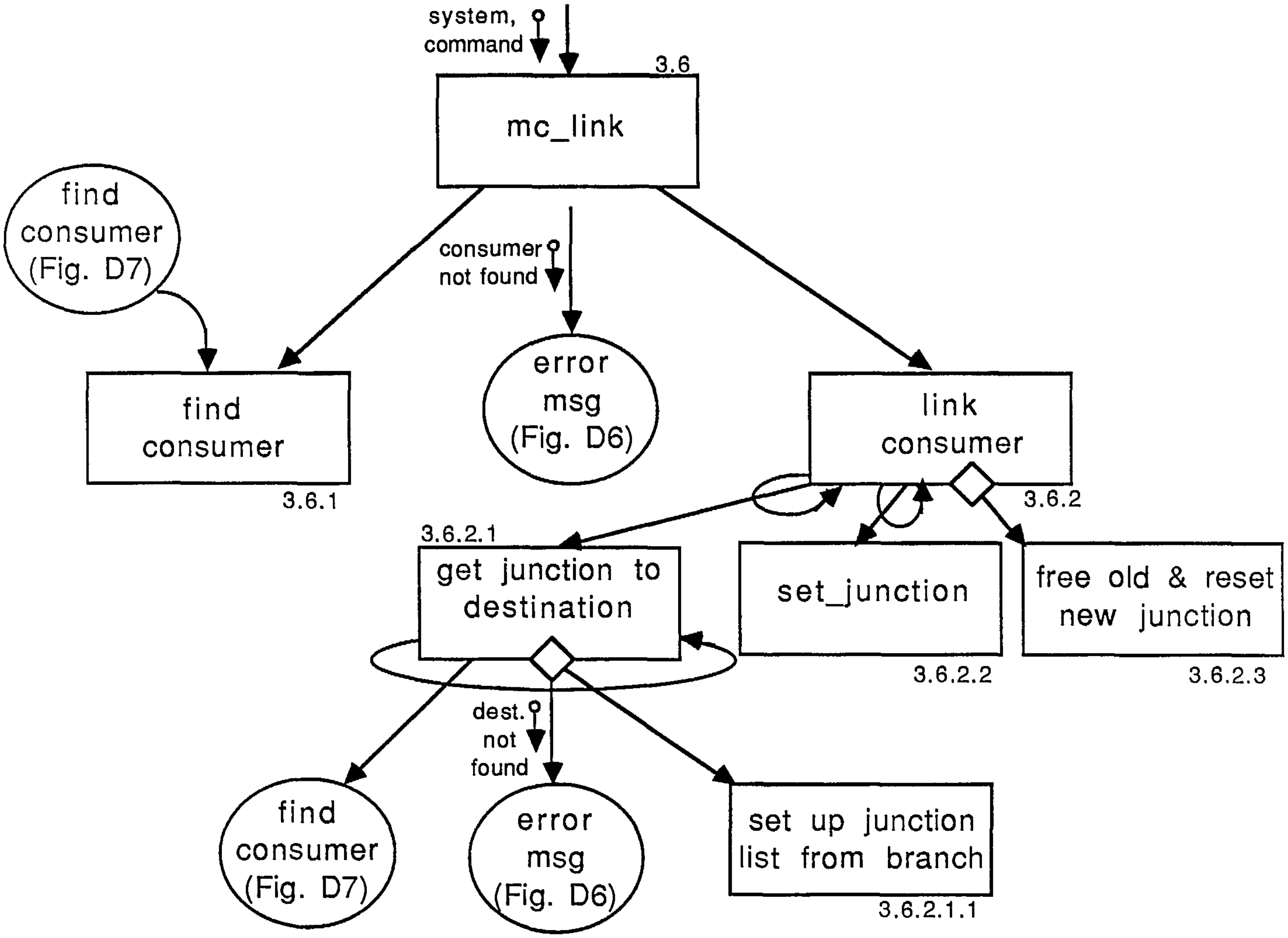


Figure D7: mc_link

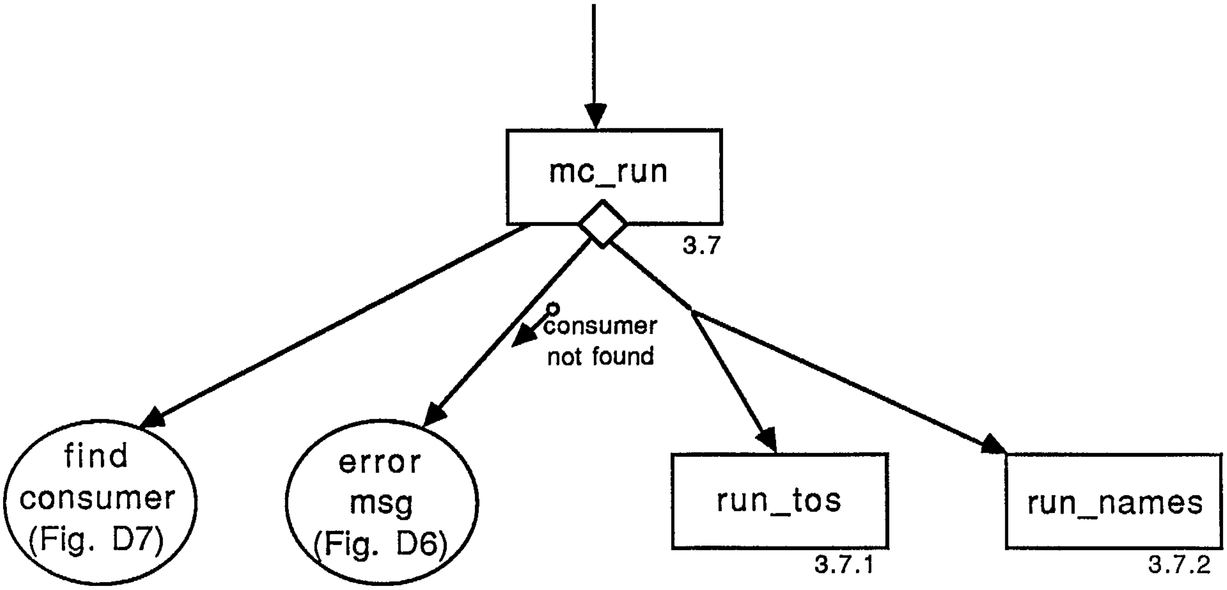


Figure D8: mc_run

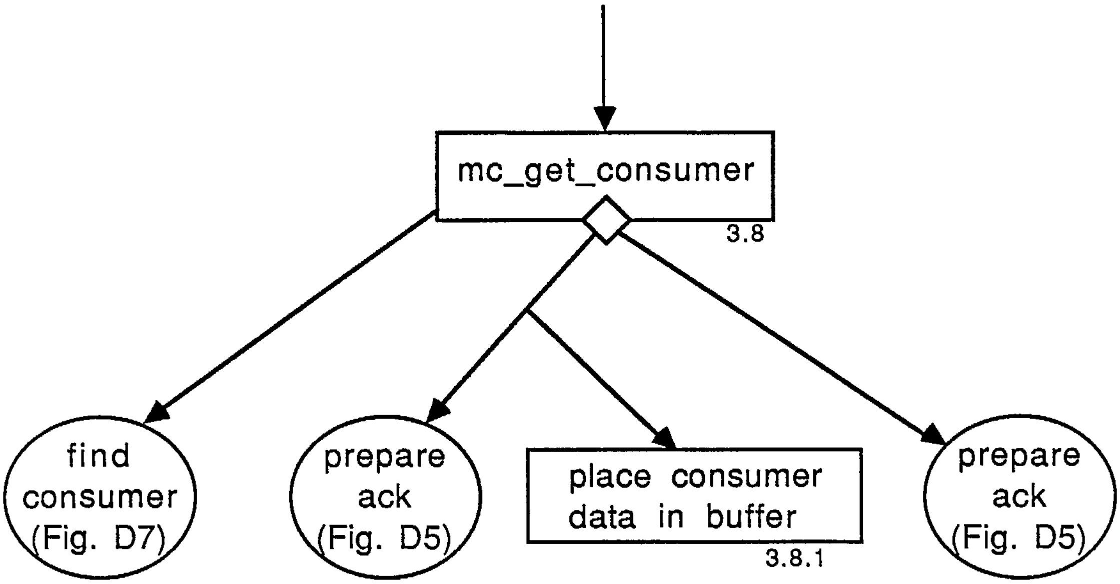


Figure D9: mc_get_consumer

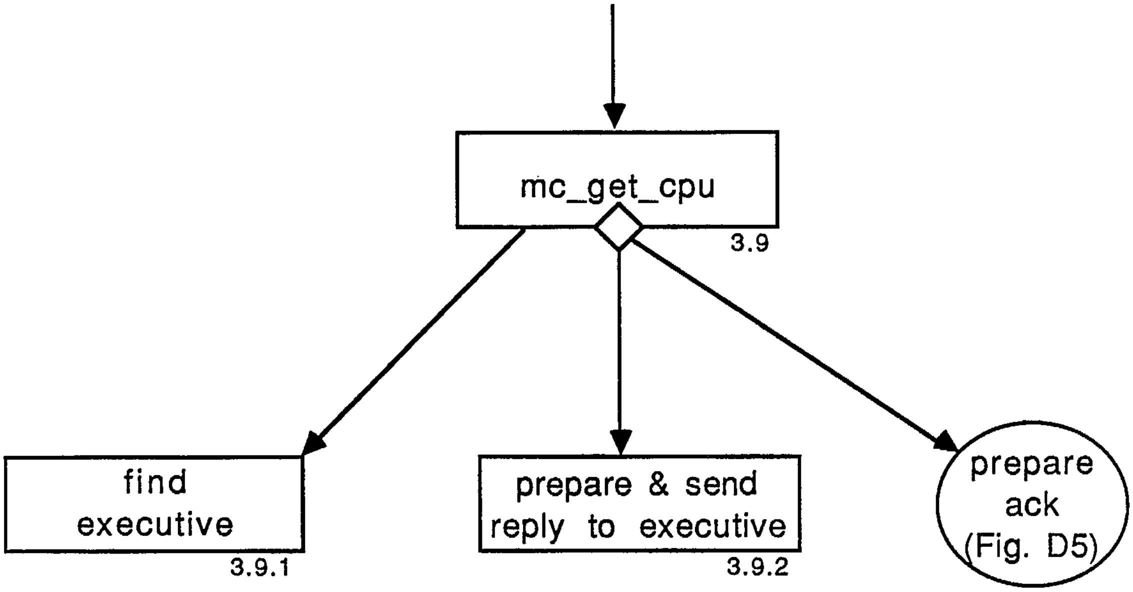


Figure D10: mc_get_cpu

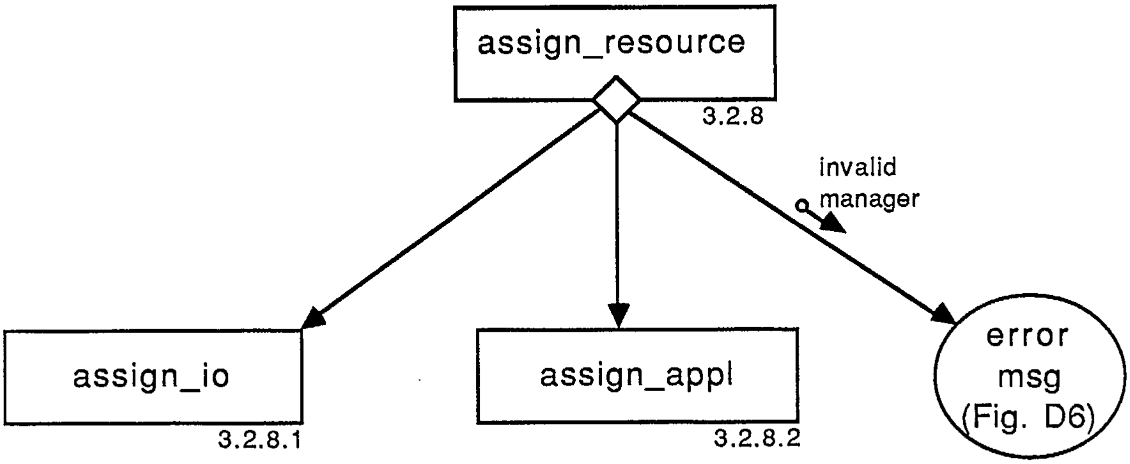


Figure D11: assign_resource

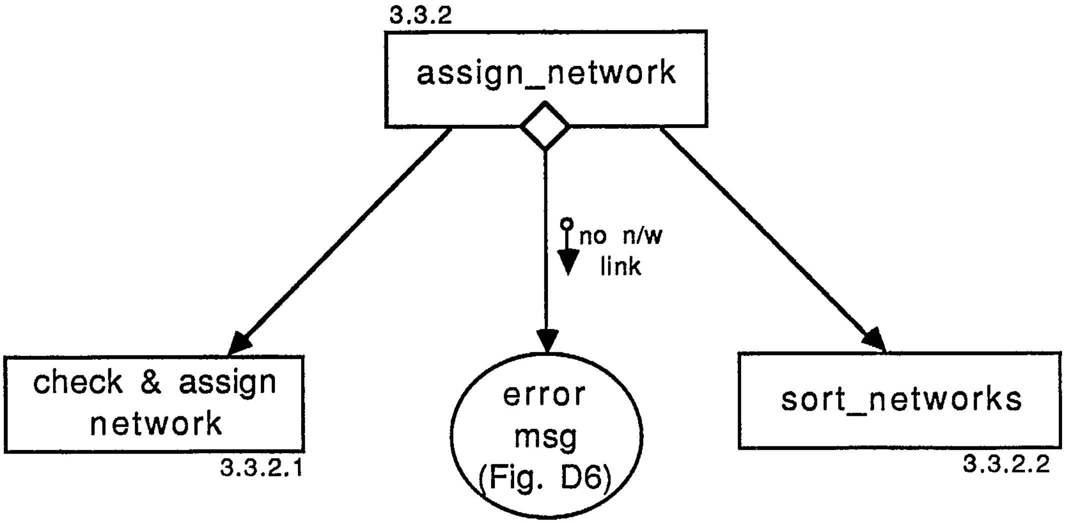


Figure D12: assign_network

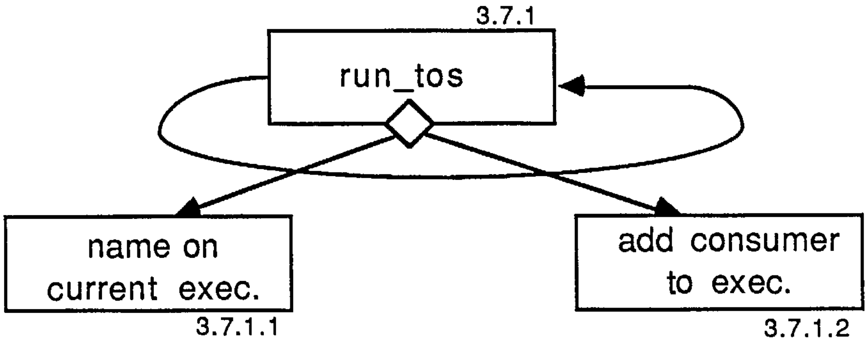


Figure D13: run_tos

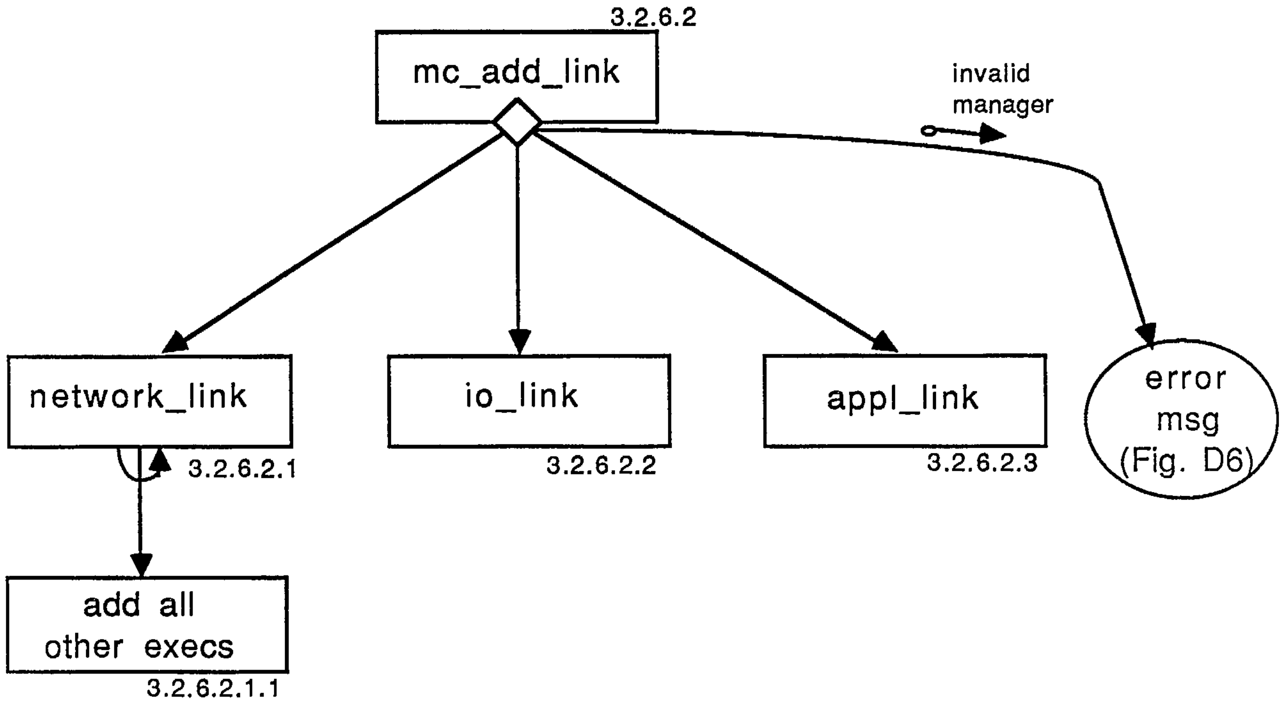


Figure D14: mc_add_link

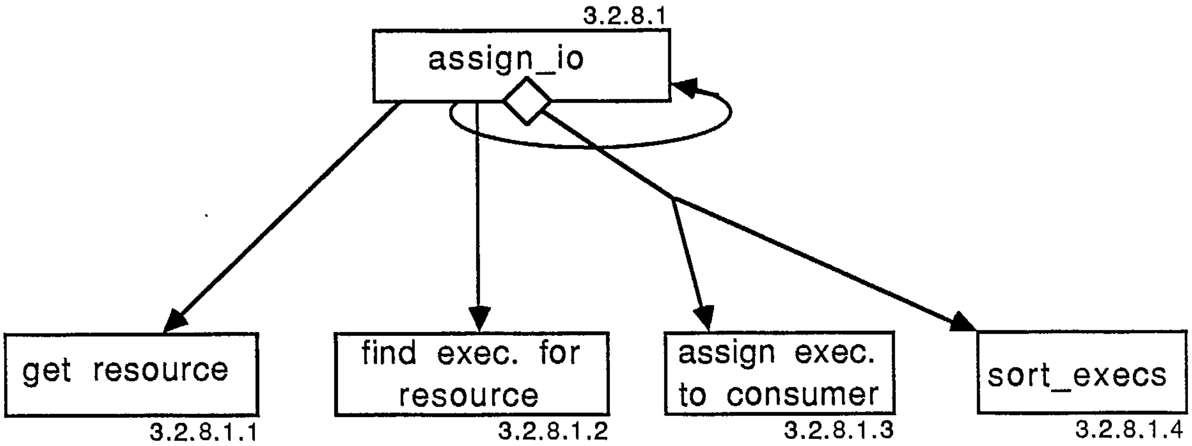


Figure D15: assign_io

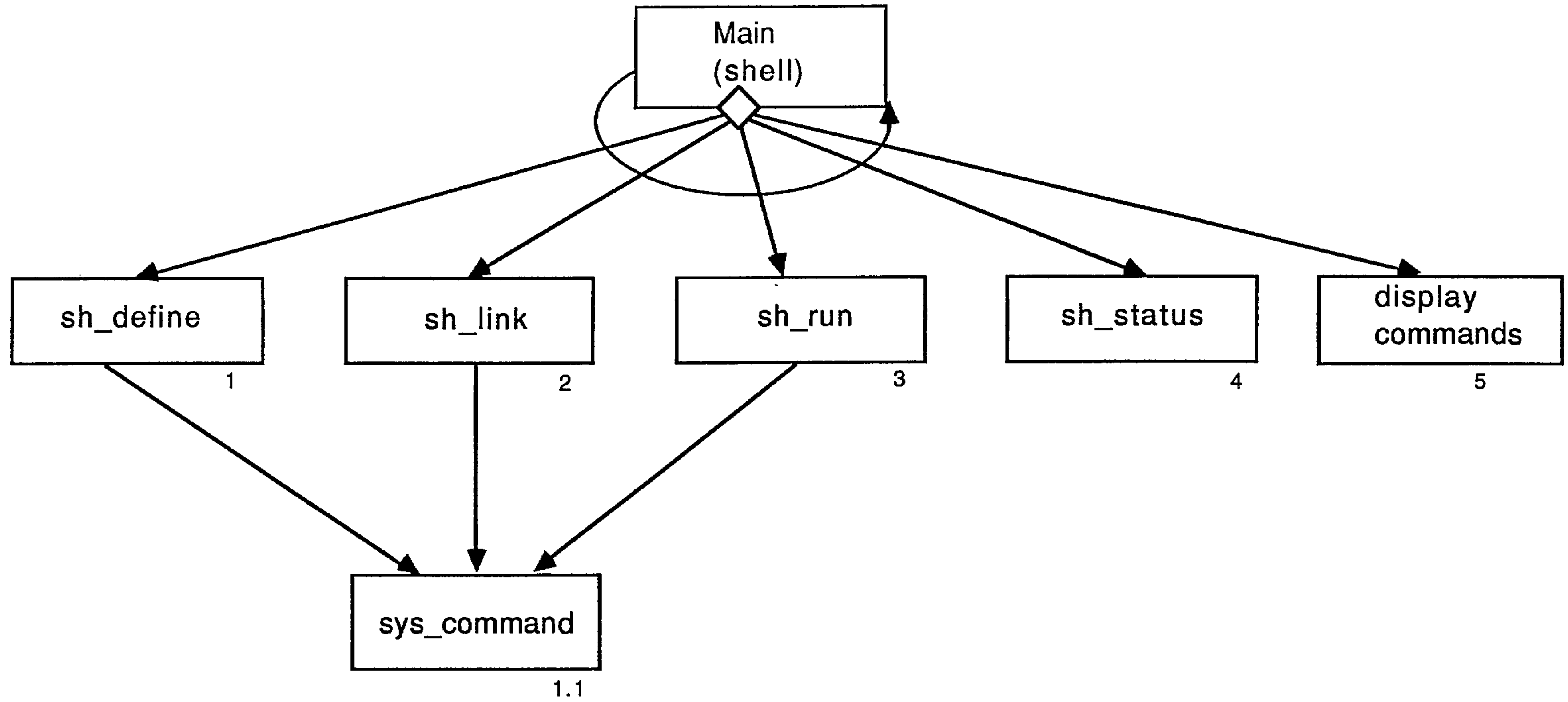


Figure H1: Main (shell)

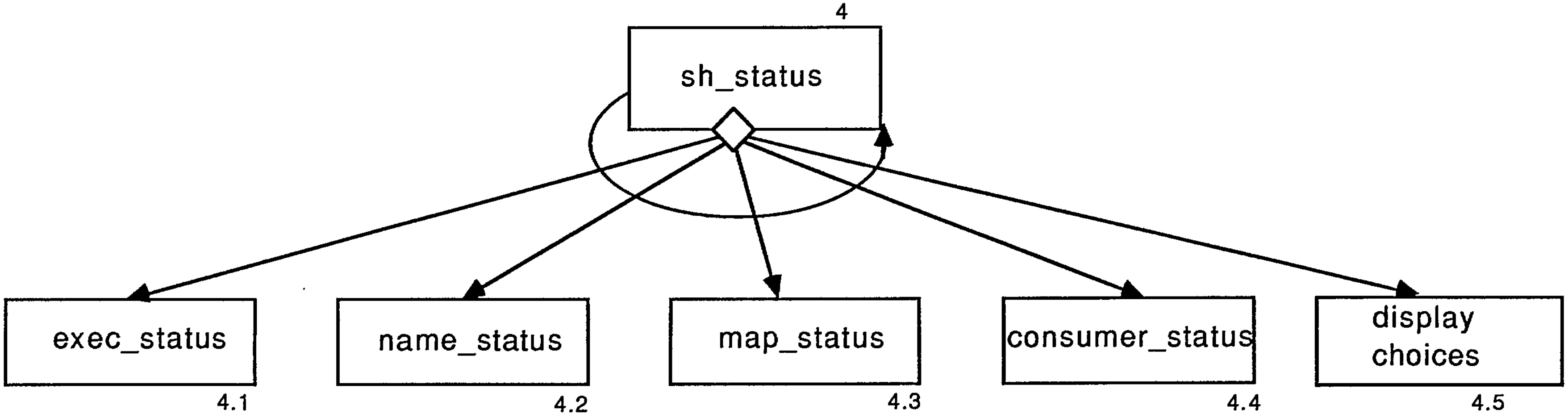


Figure H2: sh_status

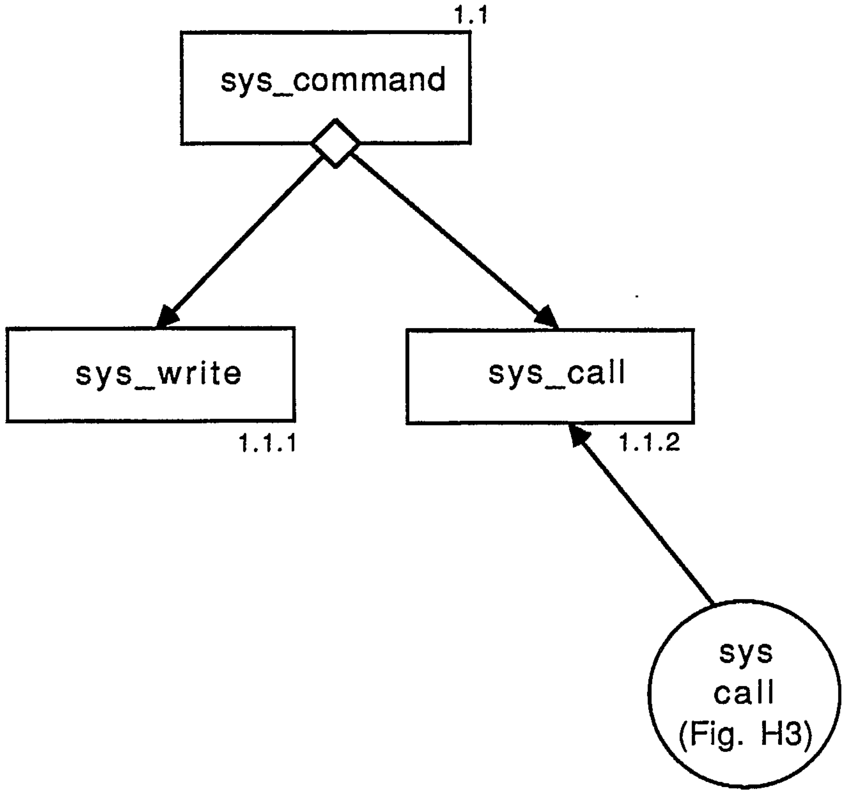


Figure H3: sys_command

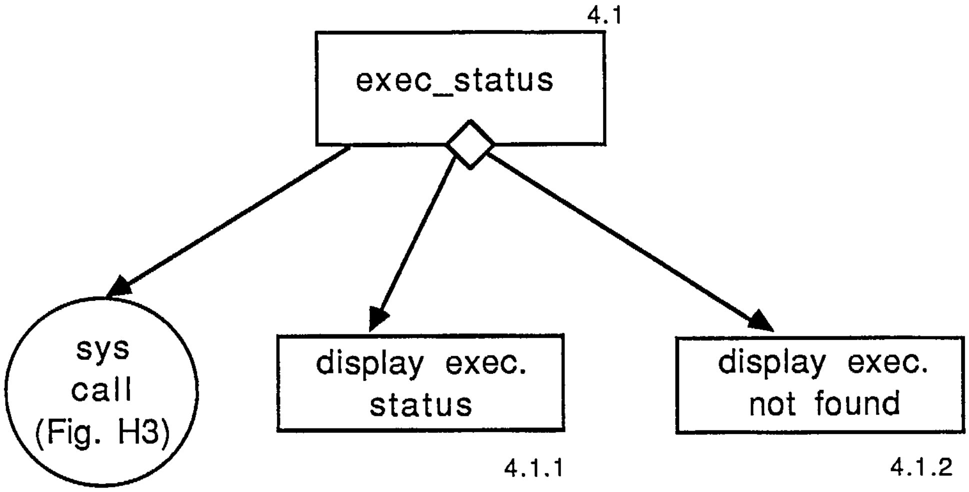


Figure H4: exec_status

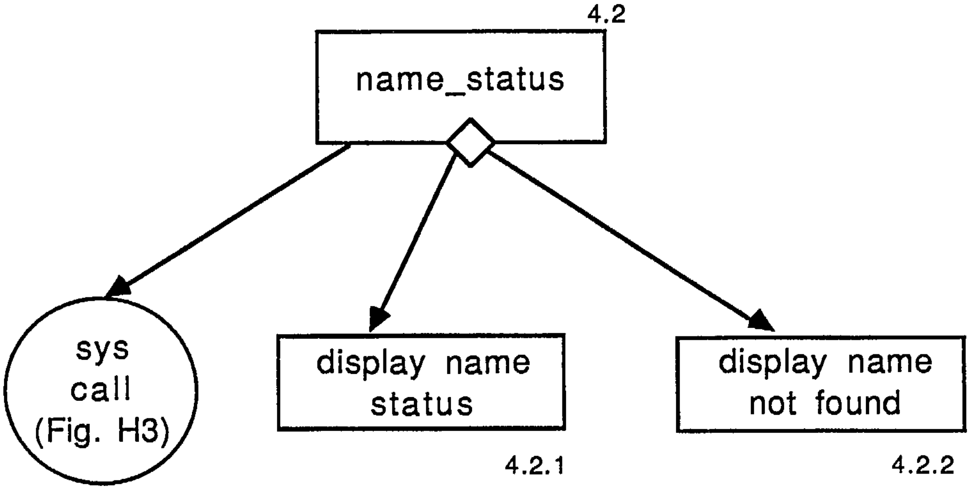


Figure H5: name_status

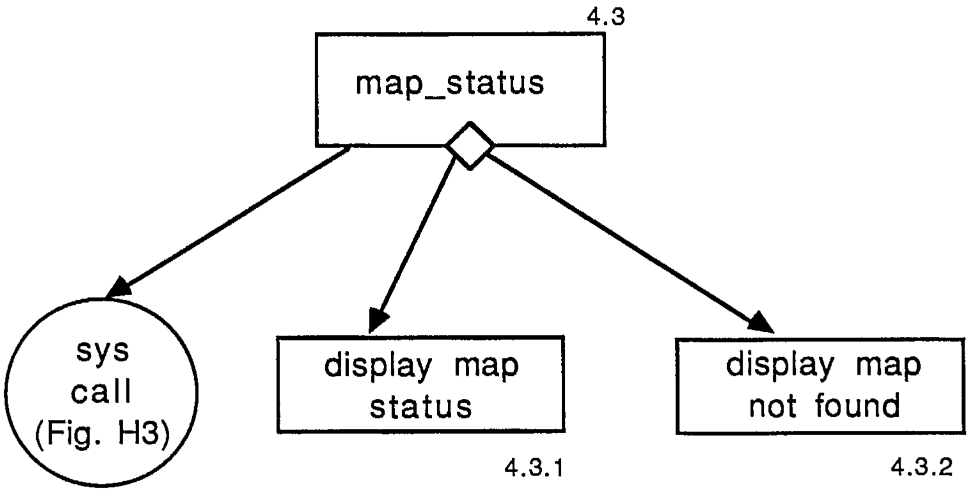


Figure H6: map_status

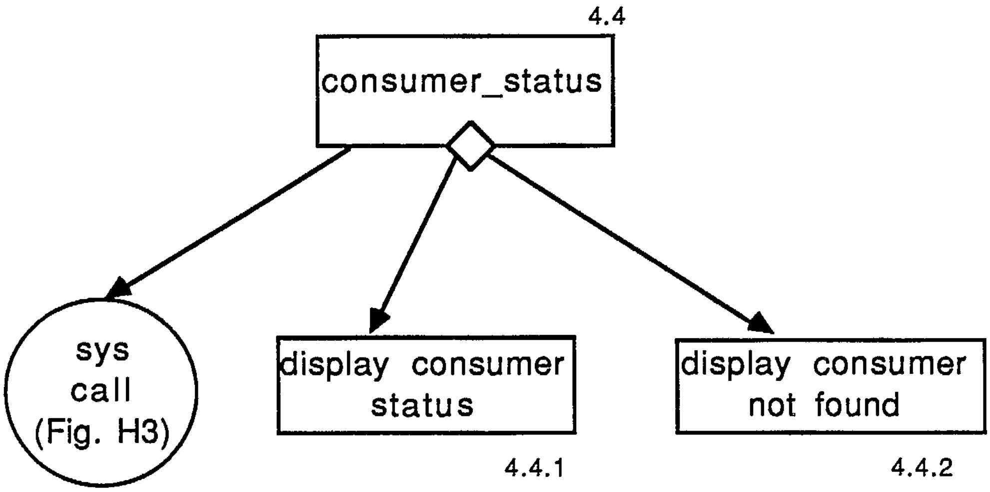


Figure H7: consumer_status

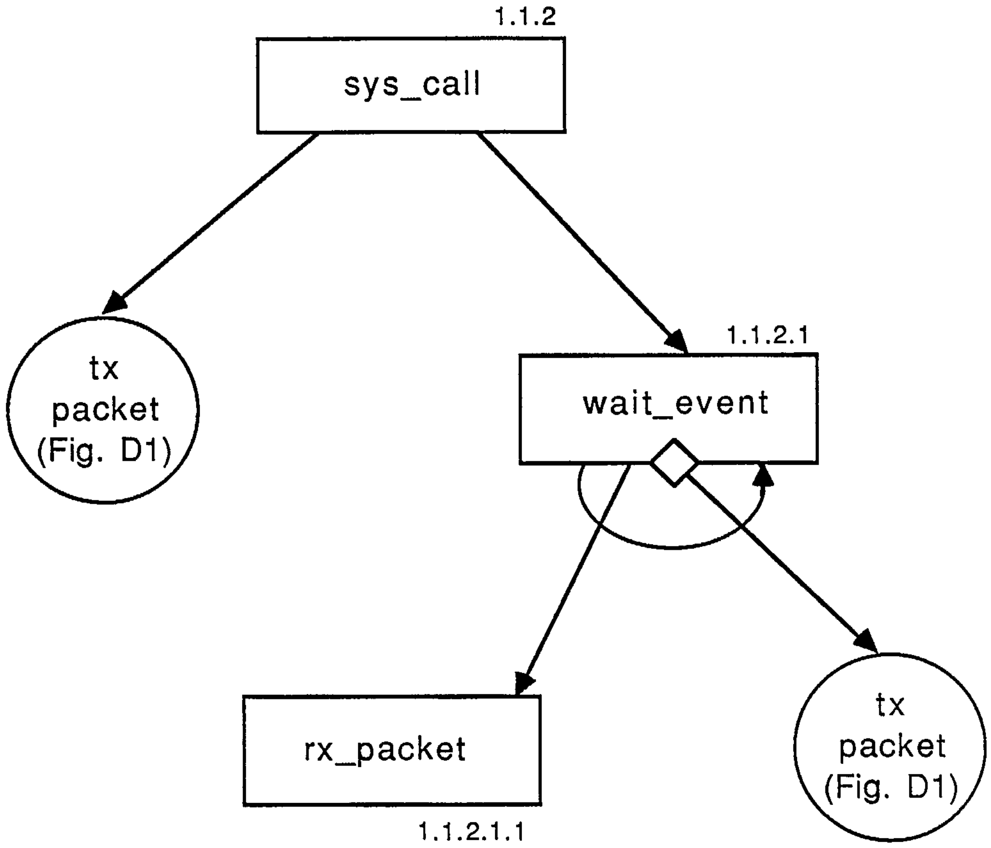


Figure H8: sys_call

Appendix C1 - CONNECTORS DRAWN IN STRUCTURE DIAGRAMS

This section lists connectors used in the structure diagrams of the simulator and the operating system described in this document. For each connector, the corresponding process name, the process number and a list of all the figures where this connector is drawn are given.

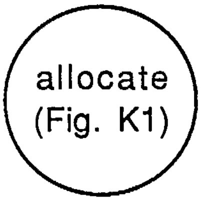
List of Connectors (Simulator)

error (Fig. S4)	Process name: Display error message Process #: 3.1 Figures: S4, S5, S8, S10, S12, S14, S15, S16
Sim. to res. (Fig. S4)	Process name: sim_to_resource Process #: 3.2 Figures: S4, S5
config (Fig. S6)	Process name: st_config Process #: 5.2 Figures: S6, S7, S8
cpu go (Figs. S7 & K1)	Process name: sim_cpu_go Process #: 6.1 (in Fig. S7) & 1 (in Fig. K1) Figures: S7, S8
free config. (Fig. S7)	Process name: st_free_config Process #: 6.2 Figures: S7, S8
cpu run (Figs. S7 & K2)	Process name: cpu_run Process #: 6.3 (in Fig. S7) & 2 (in Fig. K2) Figures: S7, S8, S12

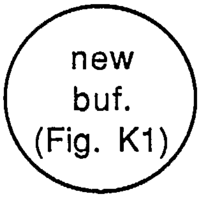
List of Connectors (Simulator) contd.

Process name: add_dsm_command
Process #: 5.3.4.1
Figures: S14

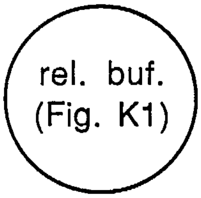
List of Connectors (O.S. Kernel)



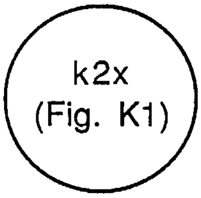
Process name: K_allocate
Process #: 1.1.1.1
Figures: K1, K4, K15.



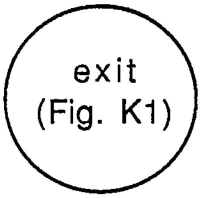
Process name: K_new_buffer
Process #: 1.1.4
Figures: K1, K9, K10, K12, K13, K15, E5



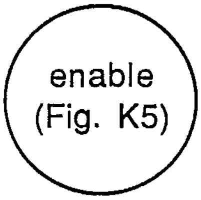
Process name: K_release_buffer
Process #: 1.1.6
Figures: K1, K9, K10, K11, K13



Process name: K_cpu_k2x
Process #: 1.1.7
Figures: K1, K9, K10, K12

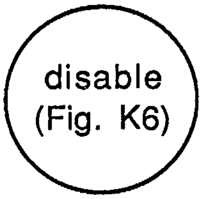


Process name: K_cpu_exit
Process #: 1.1.8
Figures: K1, K2

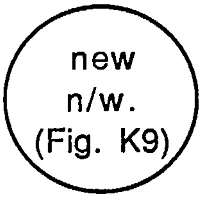


Process name: K86_enable
Process #: 1.1.8.1
Figures: K5, K6, K8

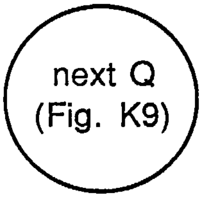
List of Connectors (O.S. Kernel) Contd.



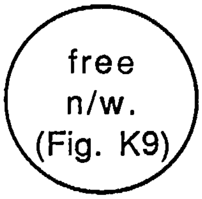
Process name: K86_disable
Process #: 1.1.8.2.2
Figures: K6, K8



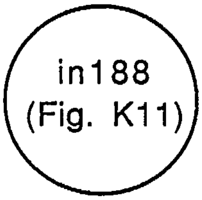
Process name: K_new_network
Process #: 1.1.8.2.1.1.1
Figures: K9, K10, K14, K15



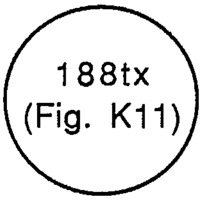
Process name: K_next_queue
Process #: 1.1.8.2.1.1.2
Figures: K9, K10



Process name: K_free_network
Process #: 1.1.8.2.1.1.3
Figures: K9, K10, E3



Process name: in_188_raw
Process #: 1.1.8.2.1.3.1.1
Figures: K11



Process name: k188_tx_packet
Process #: 1.1.8.2.1.3.1.2
Figures: K11, K13

List of Connectors (O.S. Kernel) Contd.

out188 (Fig. K11)	Process name: out_188_raw Process #: 1.1.8.2.1.3.1.3 Figures: K11, K14
free io (Fig. K11)	Process name: K_free_io Process #: 1.1.8.2.1.3.2.1 Figures: K11, E4
new io (Fig. K12)	Process name: K_new_io Process #: 1.1.8.2.1.3.1.1.1 Figures: K12, K14, K15
copy buf. (Fig. K14)	Process name: K_copy_buffer Process #: 3.1.1 Figures: K14
add Q (Fig. K14)	Process name: K_add_queue Process #: 3.1.2 Figures: K14
link control (Fig. K15)	Process name: K_link_control Process #: 4.1.1 Figures: K15

List of Connectors (O.S. Executive)

new msg.
(Fig. E3)

Process name: X_new_message
Process #: 1.1.1
Figures: E3, E4, E5, E6, E14, E17, E24, E26, E40

exec.
router out
(Fig. E3)

Process name: XR_out
Process #: 1.1.2
Figures: E3, E20, E39

next Q
(Fig. E4)

Process name: X_next_queue
Process #: 1.2.1
Figures: E4, E12, E16, E18, E28, E32, E33, E42

exec.
router in
(Fig. E4)

Process name: XR_in
Process #: 1.2.2
Figures: E4, E5, E14, E17, E24, E25, E26, E28, E40

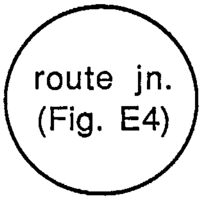
free io
(Fig. E4)

Process name: X_free_io
Process #: 1.2.3
Figures: E4, E28

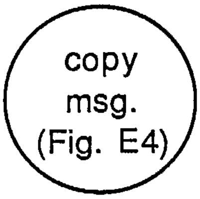
add Q
(Fig. E4)

Process name: X_add_queue
Process #: 1.2.5
Figures: E4, E10, E12, E16, E24, E26, E28, E32, E34, E35, E36, E42

List of Connectors (O.S. Executive) Contd.



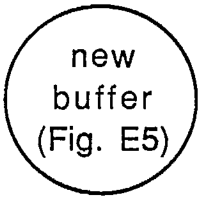
Process name: X_route_junction
Process #: 1.2.4
Figures: E4, E17, E26



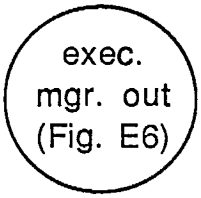
Process name: X_copy_message
Process #: 1.2.4.1
Figures: E4, E25, E39, E41

Process name: process_ready
Process #: 1.3.7
Figures: E5, E15, E16

Process name: X_report_error
Process #: 1.3.8
Figures: E5, E10, E17, E21, E25, E26, E28,
E29, E30, E31, E37, E38, E42

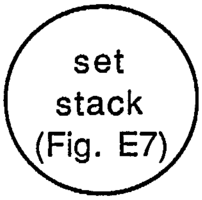


Process name: X_new_buffer
Process #: 1.3.8.1
Figures: E5, E10, E40

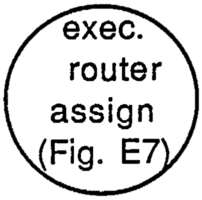


Process name: XM_out
Process #: 1.4.1
Figures: E6, E10, E41.

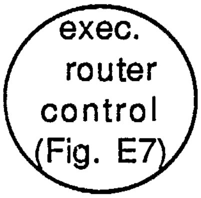
List of Connectors (O.S. Executive) Contd.



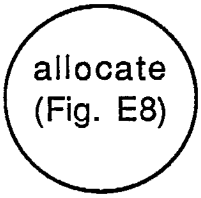
Process name: X_set_stack
Process #: 2.4.1
Figures: E7, E8



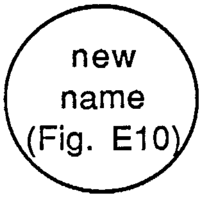
Process name: XR_assign
Process #: 2.4.2
Figures: E7, E25



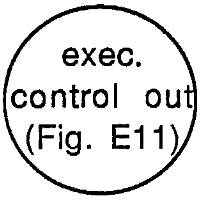
Process name: XR_control
Process #: 2.4.3
Figures: E7, E25



Process name: X_allocate
Process #: 2.5.1
Figures: E8, E9, E20, E22, E37, E38

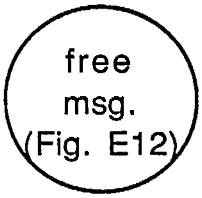


Process name: new_name
Process #: 1.1.2.1
Figures: E10, E20

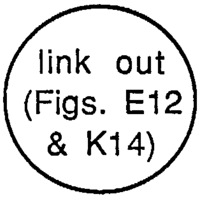


Process name: xcontrol_out
Process #: 1.2.2.1
Figures: E11, E19

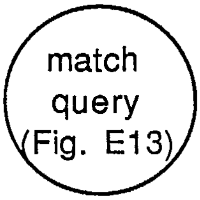
List of Connectors (O.S. Executive) Contd.



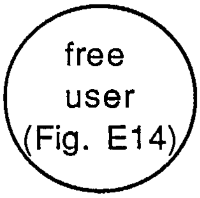
Process name: X_free_message
Process #: 1.3.1.1
Figures: E12, E16, E25, E26, E27, E28, E34, E35, E36, E42



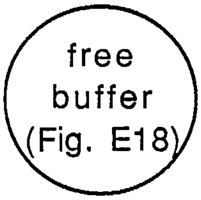
Process name: K_link_out
Process #: 1.3.1.2 (in Fig. E12) & 3 (in Fig. K14)
Figures: E12, E13, E14, E17, E18, E27, E28, E33



Process name: match_query
Process #: 1.3.2.1
Figures: E13, E15



Process name: X_free_user
Process #: 1.3.3.1
Figures: E14, E17, E18, E33



Process name: X_free_buffer
Process #: 1.3.7.1
Figures: E18, E25, E27, E28, E33, E42

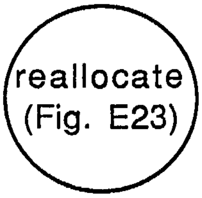
Process name: next_process
Process #: 1.3.7.2.1
Figures: E18, E34

List of Connectors (O.S. Executive) Contd.

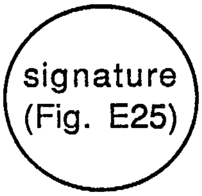
Process name: XM_assign
Process #: 2.4.2.2
Figures: E20, E21

Process name: XM_control
Process #: 2.6.1.1
Figures: E21, E22, E30, E37, E38

Process name: K_link_assign
Process #: 2.6.1.2 (in Fig. E22) & 4 (in Fig. K15)
Figures: E22, E37, E38



Process name: X_reallocate
Process #: 1.1.2.1.1
Figures: E20, E23, E31, E37, E38



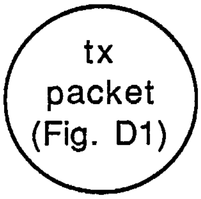
Process name: X_find_signature
Process #: 1.2.2.1.1.1
Figures: E25, E26, E35

Process name: X_command
Process #: 1.2.2.1.2.1
Figures: E25

List of Connectors (O.S. Executive) Contd.

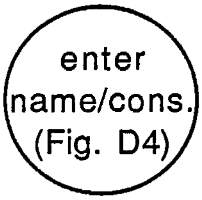
nmr done (Fig. E32)	Process name: xnmr_done Process #: 1.1.2.4.2.4 Figures: E32, E41
nmr ready (Fig. E32)	Process name: xnmr_ready Process #: 1.1.2.4.2.1.1 Figures: E32
wait 2ready (Fig. E34)	Process name: process_wait2ready Process #: 1.4.1.3.1.1 Figures: E34, E35, E36, E38

List of Connectors (O.S. DSM & shell)



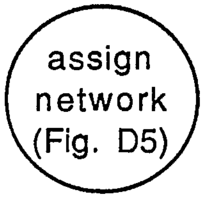
Process name: tx_packet
Process #: 1.2
Figures: D1, H8

Process name: mc_list
Process #: 3.1
Figures: D3



Process name: enter_name & enter_consumer
Process #: 3.2.7
Figures: D4, D6

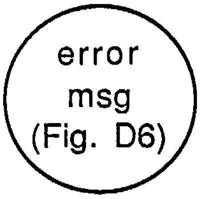
Process name: assign_resource
Process #: 3.2.8
Figures: D4, D6



Process name: assign_network
Process #: 3.3.2
Figures: D5, D6

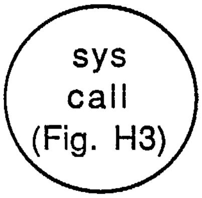
Process name: Prepare Acknowledgement
Process #: 3.3.3
Figures: D5, D9, D10

List of Connectors (O.S. DSM & shell) contd.



Process name: Display Error Message
Process #: 3.5.2
Figures: D6, D7, D8, D11, D12, D14

Process name: find consumer
Process #: 3.6.1
Figures: D7, D8, D9



Process name: sys_call
Process #: 1.1.2
Figures: H3, H4, H5, H6, H7

APPENDIX D

DATA STRUCTURES

USED BY

THE FTDCS SIMULATOR & THE OPERATING SYSTEM

Table of Contents

page

1. INTRODUCTION

This Appendix describes the data structures that are used by the FTDCS simulator and the operating system. It is a data dictionary of the data items contained within the system.

The following section lists the data structures used by FTDCS. And Appendix D1 lists the "C" function files which describe them.

2. DATA STRUCTURES

The subsections below give a list of the data structures used by FTDCS. However, the various software modules of the FTDCS simulator and the operating system do not use them as they are. They create copies of the data structures and manipulate the contents as desired. Thus, each copy of the data structure exists within the scope of that module.

At the beginning of each subsection, is a general description of data structures explained in that subsection. Appendix C will help understand the context of use for some of the data structures. For example, the kernel boot data structure "KM_Boot" in Section 2. is used by the kernel boot process "KB_boot", explained in Section 4.1 of Appendix C. However, note that some of the data structures are quite general (e.g., "Message") and that they are used in a number of places throughout the system.

For each data structure described, the following information is given:

- the purpose of having this data structure in the system (in some cases this description may be vague or it may be missing, as very little documentation about the corresponding data structure is available).
- the fields (or contents) of the data structure. Note that the fields of some data structures may be used to hold pointers to other data structures or pointers to executable functions, in which case this is specified.

2.1 Simulator Model Data

The following data structures are used by the simulator to define the model system. These data structures contain the model information such as, the processor data (e.g., handlers, links, CPUs), manager, resource, router and server data, data structure for messages passed, simulator and system data.

Simulator_Def

purpose: a data structure for the simulator (to initialize/free it).
fields: name,
init (ptr. to function which initializes simulator),
free (ptr. to function which frees simulator).

1. MDL_Handler

purpose: processor component structure for the handler.
fields: id, (handler id)
cfg_id,
cfg_count,
unit_base,
unit_count,
trap (interrupt trap routine for handler),
name.

2. MDL_Link:

purpose: processor component structure for the link.
fields: id, (link id)
cfg_id,
sim_unit,
server_id, handler_id, (server & handler ids)
unit_id,
length,
cfg_data.

3. MDL_Cpu

purpose: processor component structure for the CPU.
fields: name,
id,
type,
link_count,
unit_count,
handler_count,
server_count,
resource_count,
config_ptr,
os_memory,
os_data,
linked_execs,
nw_depth,
nw_offset,

nw_count
sys.

4. MDL_Manager

purpose: resource manager component structure.
fields: id,
link_count,
name,
cfg_count,
sys.

5. MDL_Router

purpose: router component structure (for simple & NMR routers).
fields: id,
name,
cfg_count,
sys.

6. MDL_Server

purpose: server component structure
fields: id,
name,
add_link,
cfg_id,
cfg_count.

7. MDL_Message

purpose: model message component structure.
fields: id,
to_link_id,
from_link_id,
cpu_id,
unit_id,
time_lag,
length (message length),
next, last (ptrs. to MDL_message),
data (message data).

8. MDL_Simulator

purpose: simulator component structure.
fields: id,
name,
server_id,
cpu_server,
setup, (ptr. to function to setup simulator)
free, (ptr. to function to free simulator)
in (ptr. to in entry point function),
out (ptr. to out entry point function),
assign (ptr. to assign entry point function),

control (ptr. to control entry point function),
data.

9. MDL_Resource

purpose:	resource (network, I/O or process) component structure
fields:	name, sim_id, id, type, manager (resource manager), link_count, cfg_id, cfg_count, cfg_base, sys.

10. MDL_System

purpose:	system structure.
fields:	sys, cpu2model, resource2model, sys_id2model, sys_id2resource, sys_id2cpu, sys_id2local_id, local_id2sys_id, linked_execs, nw_map_limit, nw_map_count, network_map, dsm_ptr.

11. Model

purpose:	system model structure
fields:	link_count, link_limit, link2cpu, link2resource, link2status, links (ptr. to MDL_Link, see 2 above), handler_limit, handlers (ptr. to MDL_Handler, see 1 above), simulator_limit, simulators (ptr. to MDL_Simulator, see 8 above), manager_limit, managers (ptr. to MDL_Manager, see 4 above), router_limit, routers (ptr. to MDL_Router, see 5 above), server_limit, servers (ptr. to MDL_Server, see 6 above), resource_count,

resource_limit,
resources (ptr. to MDL_Resource, see 9 above),
cpu_count,
cpu_limit,
cpus (ptr. to MDL_Cpu, see 3 above),
system (ptr. to MDL_System, see 10 above),
message_stack,
buffer_stack,
head, tail (both ptrs. to MDL_Message, see 7 above),
id_limit,
next_id,
id2link,
id2sim,
id2local.

2.2 Simulator Link Data

The following are the link data structures for the simulator, network, I188, 8086 processor etc.

1. Sim_link

purpose:	simulator link data structure.
fields:	model_id.

2. Sim_NW_Link

purpose:	simulator network link data structure.
fields:	model_id, address, link_count, NW_add0, NW_add1, NW_add2 (network address).

3. NW_Link

purpose:	network link data structure.
fields:	segment, address, offset, link_count, NW_add0, NW_add1, NW_add2 (network address).

4. MB_Link

purpose:	Multibus link data structure.
fields:	segment, address, offset.

5. I188_Link

purpose:	I188 link data structure.
fields:	segment, offset.

6. P86_Link

purpose:	8086 processor link data structure.
fields:	cs_value, cs_length, ds_value, ds_length, ss_value, ss_length.

2.3 Operating System Data

The following data structure contains the operating system data and pointers to operating system boot and initialization functions.

OS_data

purpose: operating system structure.
fields: system_id,
kernel_boot, (ptr. to kernel boot function)
km_count,
km_inits, (table of kernel initialization functions)
km_data,
kernel_entries,
handler_inits, (table of handler initialization functions)
server_inits, (table of server initialization functions)
exec_boot, (ptr. to executive boot function)
exec_ptr,
xc_count,
xc_inits, (table of executive controller initialization functions)
xm_inits, (table of executive manager initialization functions)
xr_inits, (table of executive router initialization functions)
config_data.

2.4 Operating System Message Structures

The data structures given below define structures for messages passed between operating system components.

1. X_Header

purpose:	a structure for the external header of the message.
fields:	source, destiny (i.e., destination), length.

2. I_Header

purpose:	a structure for the internal header of the message.
fields:	type (MSG_SEND, MSG_REPLY, MSG_QUERY, etc.), signature, source, destiny (i.e., destination), length.

3. Message

purpose:	a data structure for the message.
fields:	internal (of type I_Header, see 2 above), data.

4. Network_Pkt

purpose:	a data structure for a network packet.
fields:	external (of type X_Header, see 1 above), internal (of type I_Header, see 2 above), data.

5. IO_Pkt

purpose:	a data structure for an I/O packet.
fields:	length, data.

6. EN_Address

purpose:	a data structure for the address of an ethernet.
fields:	add0, add1, add2.

7. EN_Data

purpose:	a data structure for the ethernet data.
fields:	external (of type X_Header, see 1 above), internal (of type I_Header, see 2 above), msgbuf (a buffer of a particular size to hold data).

8. EN_Header

purpose: a data structure for the ethernet data header.
fields: source (of type EN_address, see 6 above),
destiny (i.e., destination) (of type EN_address, see 6 above),
unused.

9. Incoming

purpose: an incoming data on ethernet.
fields: header (of type EN_Header, see 8 above),
data (of type EN_Data, see 7 above).

10. Ethernet_Pkt

purpose: a data structure for an ethernet packet.
fields: header (of type EN_Header, see 8 above),
external (of type X_Header, see 1 above),
internal (of type I_Header, see 2 above),
data.

11. User_Pkt

purpose: a data structure for the user packet.
fields: message,
location,
notify (ptr. to a notify function),
parameter,
reply_length,
reply,
status,
return_length.

12. Name

fields: id, local_id, unit_id.

13. Link

purpose: a link data structure.
fields: server_id, unit_id, length.

2.5 Kernel Data Types

The data structures given below hold kernel specific information, including the kernel processor, link, server, and memory data.

1. K_Event

purpose: a structure to hold kernel events.
fields: next (ptr. to K_Event),
unit_id,
data.

2. X_Event

purpose: a data structure to hold executive events.
fields: next (ptr. to X_Event),
xid,
data.

3. KM_Cpu

purpose: kernel processor manager data structure.
fields: active_user,
user_mode, idle_mode,
kevent_limit,
kernel_pending,
kernel_busy,
kernel_pending_head (ptr. to K_Event, see 1 above),
kernel_pending_tail (ptr. to K_Event, see 1 above),
kernel_next_head (ptr. to K_Event, see 1 above),
kernel_next_tail (ptr. to K_Event, see 1 above),
xevent_limit,
exec_pending,
exec_busy,
exec_pending_head (ptr. to X_Event, see 2 above),
exec_pending_tail (ptr. to X_Event, see 2 above),
exec_next_head (ptr. to X_Event, see 2 above),
exec_next_tail (ptr. to X_Event, see 2 above),
user_busy.

4. K_Server

purpose: kernel server data structure.
fields: server_id,
link_count,
control, assign, in, out (ptrs. to entry point functions),
data.

5. KM_Link

purpose: kernel link manager data structure.
fields: unit_count,
unit2kid,
server_count,
servers (ptr. to K_Server, see 4 above),

link_count,
link_size,
link_table (ptr. to Link, see 13 in Section 2.4),
links,
kid_limit, kid_expand, kid_count,
next_kid,
kid2server, kid2local.

6. KM_Memory

purpose: kernel memory manager data structure.
fields: memory,
buffers,
en_packets,
network_packets,
io_packets,
user_packets,
queues.

7. Context

purpose: context information data structure.
fields: cs_value, cs_length,
ds_value, ds_length,
ss_value, ss_length,
sp_value.

2.6 Kernel/Executive Boot and Initialization Functions

Six table structures are described below. They contain the functions used for the following: kernel boot, kernel handler initialization, kernel server initialization, executive boot, executive resource manager initialization and executive routing algorithm initialization.

1. KM_Boot

purpose:	a table of kernel boot functions.
fields:	KMB_memory (kernel memory manager boot function), KMB_cpu (kernel processor manager boot function), KMB_link (kernel link manager boot function).

2. KH_Inits

purpose:	a table of kernel handler initialization functions.
fields:	KHI_bus (bus handler initialization function), KHI_enet (enet handler initialization function), KHI_device (device handler initialization function), KHI_process (process handler initialization function).

3. KS_Inits

purpose:	a table of kernel server initialization functions.
fields:	KSI_bus (bus server initialization function), KSI_enet (enet server initialization function), KSI_device (device server initialization function), KSI_process (process server initialization function).

4. XCB_Boot

purpose:	a table of executive boot functions.
fields:	XCB_router (executive router boot function), XCB_manager (executive resource manager boot function), XCB_control (executive controller boot function).

5. XM_Inits

purpose:	a table of executive resource manager initialization functions.
fields:	XMI_network (executive network manager initialization function), XMI_io (executive I/O manager initialization function), XMI_process (executive application process manager initialization function).

6. XR_Inits

purpose:	a table of routing algorithm initialization functions.
fields:	XRI_simple (simple routing algorithm initialization function), XRI_nmr (NMR routing algorithm initialization function).

2.7 Kernel Handler and Server Data

This section gives the data structures for the kernel handler and server tables.

1. **Handler_Def**

purpose:	kernel handler data structure for the handler table (below).
fields:	name, trap (interrupt service routine for the handler).

2. **handler_defs**

purpose:	a table of handlers (each handler structure as described in 1 above).

3. **Server_Def**

purpose:	kernel server data structure for the server table (below).
fields:	name, add_link.

4. **server_defs**

purpose:	a table of servers (each server structure as described in 3).

2.8 Kernel/Executive Configuration Data

The following data structures contain the definitions for the hardware independent data in the kernel/executive configuration data blocks.

1. CD_Header

purpose: configuration data header.
fields: code,
cpu_id,
kernel_length,
executive_length,
dsm_length,
config_length.

2. KCD_Manager

purpose: kernel manager configuration data.
fields: length.

3. KCD_Memory

purpose: kernel memory manager configuration data.
fields: header (ptr. to KCD_Manager, see 2 above),
exec_id,
memory_size,
min_buffer,
max_buffer.

4. KCD_Cpu

purpose: kernel processor manager configuration data.
fields: header (ptr. to KCD_Manager, see 2 above),
kevent_limit,
xevent_limit.

5. KCD_Link

purpose: kernel link manager configuration data.
fields: header (ptr. to KCD_Manager, see 2 above),
initial_kid,
expand_kid,
handler_count,
unit_count,
server_count,
link_count,
link_size.

6. KCD_Handler

purpose: kernel handler configuration data.
fields: unit_count, vector.

7. KCD_Server

purpose:	kernel server configuration data.
fields:	link_count.

8. XCD_Executive

purpose:	executive configuration data block.
fields:	exec_id, memory_size, initial_messages, expand_messages, initial_queue, expand_queue, initial_xid, expand_xid, initial_names, expand_names, initial_consumers, expand_consumers, router_count, linked_execs, dsm_id, dsm_resource, dsm_name_count, dsm_router, dsm_name, manager_count, resource_count, link_count.

9. XCD_Resource

purpose:	executive resource configuration data block.
fields:	resource_id, manager_id, type, sys_link_count, link_count.

10. XCD_Consumer

purpose:	executive consumer configuration data block.
fields:	consumer_id, router_type, name_count.

2.9 Executive Data

The data structures given below contain information for the executive and its components.

1. X_DSM_Query

purpose:	data structure to hold the query message to the DSM.
fields:	message, notify, parameter.

2. XC_Control

purpose:	executive controller structure.
fields:	exec_id, linked_execs, dsm_id, dsm_resource, dsm_name_count, dsm_data, dsm_name, signature, dsm_query_stack, dsm_query_queue, in, out (in & out entry point function pointers).

3. X_Resource

purpose:	executive resource data structure.
fields:	system_id, sys_link_count, manager_id, resource_id, type, link_base, link_count.

4. X_Manager

purpose:	executive manager data structure.
fields:	system_id, sys_link_count, manager_id, resource_base, resource_count, link_base, link_count, in, out, assign, control (all entry point functions), data.

5. XM_Control

purpose:	the complete executive manager data structure.
fields:	manager_count, managers (ptr. to X_Manager, see 4 above), link_count, resource_count,

resources (ptr. to X_Resource, see 3 above),
xid_limit,
xid_expand,
xid_count,
next_xid,
xid2manager,
xid2local,
in, out, assign, control (ptrs. to entry point functions).

6. X_Name

purpose: name data structure.
fields: consumer_id,
consumer_link,
xid,
state.

7. X_Consumer

purpose: consumer data structure.
fields: consumer_id,
router,
name_count,
name_list,
in_pending,
out_pending,
data.

8. X_Router

purpose: router data structure.
fields: router_id,
data,
in, out, assign, control (ptrs. to entry point functions).

9. XR_Control

purpose: the complete executive router structure.
fields: consumer_count,
consumer_limit,
consumer_stack,
consumers (ptr. to X_consumer, see 7 above),
name_count,
name_limit,
name_stack,
names (ptr. to X_Name, see 6 above),
in_pending_count,
in_pending,
out_pending_count,
out_pending,
router_count,
routers (ptr. to X_Router, see 8 above),
in, out, assign, control (ptrs. to entry point functions).

10. Executive

purpose:	the complete executive data structure.
fields:	exec_id, memory, messages, queues, exec_control (ptr. to XC_Control, see 2 above), manager_control (ptr. to XM_Control, see 5 above), router_control (ptr. to XR_control, see 9 above).

2.10 Executive Commands Data

The following data structures contain information about the executive commands. They contain pointers to the actual command function in addition to other related data.

1. **XC_Add_Consumer**

purpose:	add consumer command data structure.
fields:	command, consumer.

2. **XC_Add_Name**

purpose:	add consumer name command data structure.
fields:	command, name.

3. **XC_Status**

purpose:	executive status command data structure.
fields:	command.

4. **XAS_Executive**

purpose:	executive memory and consumer information data structure.
fields:	memory_f_bytes, memory_f_blocks, memory_u_bytes, memory_u_blocks, message_count, free_messages, consumer_count, consumer_limit, name_count, name_limit, xid_count, xid_limit.

5. **XAS_Kernel**

purpose:	kernel memory information data structure.
fields:	memory_f_bytes, memory_f_blocks, memory_u_bytes, memory_u_blocks, user_count, free_users, io_count, free_ios, network_count, free_networks, en_count, free_ens, buffer_count, free_buffers, buffer_bytes, buffer_free_bytes.

6. **XA_Status**

purpose:	status information for executive and kernel information.
fields:	exec_id, executive (of type XAS_Executive, see 4 above), kernel (of type XAS_Kernel, see 5 above),

7. **XC_Set_Junction**

purpose:	set junction command data structure.
fields:	command, name_id.

2.11 DSM Commands Data

Each of the following data structures shown below contain pointers to the DSM command functions in addition to other related data.

1. **MC_boot**

   purpose: data structure for the DSM boot command.
   fields: command (ptr. to command function).

2. **MC_Unknown_Con**

   purpose: data structure for unknown consumer DSM command.
   fields: command (ptr. to command function),
   name_id,
   exec_id.

3. **MC_Unknown_Name**

   purpose: data structure for unknown name DSM command.
   fields: command (ptr. to command function),
   name_id,
   exec_id.

4. **MC_Define**

   purpose: data structure for DSM define command.
   fields: command (ptr. to command function),
   name,
   resource,
   router_type,
   name_count.

5. **MC_Link**

   purpose: data structure for DSM link command.
   fields: command (ptr. to command function),
   name,
   junction.

6. **MCD_Junction**

   purpose: data structure for the junction.
   fields: branch_id, name.

7. **MC_Run**

   purpose: data structure for DSM run command.
   fields: command (ptr. to command function), name.

8. **MC_Get_Resource**

   purpose: data structure for DSM get resource command
   fields: command (ptr. to command function), name.

9. MA_Get_Resource

fields: ack,
resource_id,
name,
manager_id,
type,
link_count,
load_count,
exec_count,
exec_mask.

10. MC_Get_Map

purpose: data structure for the get map DSM command.
fields: command (ptr. to command function), exec_name.

11. MA_Get_Map

fields: ack,
exec_name,
map_count.

12. MAD_Map_Entry

fields: to_name,
depth,
count.

13. MAD_Link

fields: id, name.

14. MC_Get_Consumer

purpose: data structure for the get consumer DSM command.
fields: command (ptr. to command function), name.

15. MA_Get_Consumer

fields: ack,
name,
consumer_id,
router_type,
name_count.

16. MC_Get_name

purpose: data structure for the get name DSM command.
fields: command (ptr. to command function), name_id.

17. MA_Get_Name

fields: ack,
name_id,
map_count,
exec_name,

resource_name,
consumer_name.

18. MAD_Name

fields: exec_name, resource_name.

19. MC_Get_Exec

purpose: data structure for the get exec. DSM command.
fields: command (ptr. to command function), name.

20. MA_Get_Exec

fields: ack, status.

21. MC_Error

purpose: data structure for the DSM error command.
fields: command (ptr. to command function),
code,
exec_id,
text.

22. MA_Error

fields: ack.

23. MC_Halt

purpose: data structure for the DSM halt command.
fields: command (ptr. to command function).

24. MC_Get_Status

purpose: data structure for the get status DSM command.
fields: command (ptr. to command function), name.

2.12 Consumer Junction Data

The following define the consumer junction data types. The data structures for the junction and the junction branch are given below.

Junction

purpose:	junction data structure.
fields:	branch_count, route_count.

Branch

purpose:	junction branch data structure.
fields:	route_base, route_count.

2.13 Utility Function Data

The following describes the utility function data structures. Copies of the utility data structures are used a number times throughout the system.

1. **IO_Buffer**

   purpose: an I/O buffer data structure.
   fields: fid,
   status,
   ptr,
   length,
   data_length,
   data.

2. **U_Buffer**

   purpose: a user buffer data structure.
   fields: size, user_id.

3. **Stack**

   purpose: stack management structure.
   fields: memory,
   items_used,
   item_count,
   item_size,
   expand_count,
   next_block,
   next_item.

4. **Q_link**

   purpose: queue linkage structure.
   fields: next (a pointer to Q_link),
   data.

5. **Queue**

   purpose: queue management structure.
   fields: head, tail (both pointers to Q_link, see 4 above).

6. **U_Memory_Block**

   purpose: memory linkage structure.
   fields: size, last_size.

7. **U_Memory_Free**

   purpose: data structure for free memory.
   fields: block (ptr. to U_Memory_Block, see 6 above),
   next_free, last_free (both pointers to U_Memory_Free).

8. U_Memory

purpose:	memory management structure.
fields:	expand_size, alloc_fn (ptr. to allocate memory function), error_fn (ptr. to display memory management errors function), used_blocks, used_bytes, free_blocks, free_bytes, first_block, free_head, free_tail (all ptrs. to U_Memory_Free, see 7 above).

9. U_Cmd_Ctl

purpose:	command control structure.
fields:	memory, buffer_limit, buffer_ptr.

10. U_Buffer_Ctl

purpose:	user buffer control structure.
fields:	memory, small_size, small_count, small_limit, next_small (ptr. to U_Buffer, see 2 above), large_size, large_limit, large_count, next_large (ptr. to U_Buffer, see 2 above), oversize_count, oversize_bytes.

2.14 Machine Data Types

The following lists the machine data types. The data fields in Sections 2.1 through 2.13, (which are not pointers to functions or other data structures) are any one of the structures given below.

Byte	8 bits unsigned.
Word	16 bits unsigned.
Long	32 bits unsigned.
Integer	16 bits integer.
Address	pointer to 8 bits.
Pointer	pointer to 16 bits.
Indirect	pointer to pointer.
Function	pointer to function returning Word.
Table	structure with Functions, (for Function see above).
OS_Channel	structure with pid (Long) & dsc (dsc$descriptor_s).
OS_Process	structure with pid (Long).
OS_File	structure of type Long (for Long, see above).

APPENDIX D1 - FILES CONTAINING FTDCS DATA STRUCTURES

The following is a list of all the "C" function files (in alphabetical order) which contain the C code for the data structures explained in Appendix D.

config.h,
config86.h,
configsim.h,
cpudef.inc,
enet.h,
executiv.h,
handlerdef.inc,
junction.h,
kernel.h,
mcommand.h,
mdlsimulator.inc,
message.h,
model.h,
os.h
serverdef.inc,
sysmc.h,
utility.h,
xcommand.h,

APPENDIX E

PERFORMANCE ANALYSIS TABLES

APPENDIX E

PERFORMANCE ANALYSIS TABLES

LIST OF TABLES

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
st_memory	1	10	8	110	660
st_system	12	288	239	2750	16500
st_file	47	1201	997	11,380	68,280
st_io	*	*	*	*	*
st_console	*	*	*	*	*
st_go	*	*	*	*	*
st_node	*	*	*	*	*

Note: The code for the functions (for which no data given in this table) is very convoluted. The total number of lines of code in these functions can only be determined at execution time.

Table 1: Main Functions of the Simulator

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
St_set_system	0	47	39	390	2340
St_sw_config	4	139	115	1270	7620
read_consumers	3	82	68	770	4620
std_define	1	23	19	220	1320
std_link	2	35	29	350	2100
std_run	1	9	8	110	660
st_sys_config	4	95	79	910	5460
add_sys_manager	0	11	9	90	540
add_sys_router	0	12	10	100	600
add_sys_resource	0	15	13	130	780
add_sys_exec	0	22	18	180	1080

Table 2: Simulator System Definition Functions

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
st_config	19	470	390	4470	26,820
st_config_header	0	11	9	90	540
st_config_exec	4	198	164	1760	10,560
config_resources	0	50	42	420	2520
config_execs	1	44	37	400	2400
add_exec_consumer	0	24	20	200	1200
config_dsm	0	65	54	540	3240
st_simconfig	17	446	370	4210	25,260
simconfig_managers	0	14	12	120	720
simconfig_handlers	0	17	14	140	840
simconfig_servers	0	16	13	130	780
simconfig_resources	0	8	7	70	420
simconfig_links	0	22	18	180	1080
simconfig_kernel	3	98	81	900	5400
simconfig_kmemory	0	13	11	110	660
simconfig_kcpu	0	11	9	90	540
simconfig_klink	3	125	104	1130	6780
sim_network_link	0	23	19	190	1140
sim_io_link	0	18	15	150	900
sim_process_link	0	19	16	160	960

Table 3: Local Configuration Specification Functions

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
stf_data	4	189	157	1690	10,140
add_header	0	31	26	260	1560
add_kernel	0	13	11	110	660
add_exec	0	94	78	780	4680
add_dsm	1	25	21	240	1440
add_dsm_command	0	18	15	150	900
stf_table	3	102	85	940	5640
tbl_header	0	28	23	230	1380
tbl_handlers	0	28	23	230	1380
tbl_servers	0	28	23	230	1380

Table 4: Simulator Configuration Support Functions.

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
sh_define	10	104	86	1160	6960
sh_link	10	123	102	1320	7920
sh_run	9	96	80	1070	6420
sys_command	6	87	72	900	5400
sh_status	18	273	227	2810	16,860
consumer_status	7	119	99	1200	7200
exec_status	7	134	111	1320	7920
sys_accept	1	18	15	180	1080
sys_query	1	25	21	240	1440
sys_reply	1	19	16	190	1140
sys_call	4	71	59	710	4260
sys_receive	4	66	55	670	4020
sys_send	1	17	14	170	1020
sys_ready	4	61	51	630	3780
wait_event	2	48	40	460	2760
setup	0	7	6	60	360
tx_packet	0	6	5	50	300
rx_packet	0	6	5	50	300

Table 5: Distributed Software Implementation Functions

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
KB_boot	143	2741	2275	27,040	162,240
KMB_cpu	1	34	28	310	1860
K_cpu_kernel	9	146	121	1480	8880
K_cpu_executive	2	35	29	350	2100
K_cpu_enter	0	10	8	80	480
K_cpu_exit	12	178	148	1840	11,040
K_cpu_fork	0	14	12	120	720
K_cpu_k2x	0	13	11	110	660
K_cpu_x2k	11	168	140	1730	10,380
K_cpu_x2u	0	8	7	70	420
K_cpu_u2x	0	6	5	50	300
KU_enable	0	4	3	30	180
KU_disable	0	4	3	30	180

Table 6: Kernel Processor Management Functions.

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
KMB_memory	1	33	27	300	1800
K_refill	0	10	8	80	480
K_mem_error	0	6	5	50	300
K_allocate	0	7	6	60	360
K_reallocate	0	10	8	80	480
K_free	0	5	4	40	240
K_set_stack	0	10	8	80	480
K_add_queue	0	6	5	50	300
K_next_queue	0	7	6	60	360
K_new_buffer	0	8	7	70	420
K_release_buffer	1	12	10	130	780
K_free_buffer	0	5	4	40	240
K_copy_buffer	0	6	5	50	300
K_new_network	0	6	5	50	300
K_free_network	0	5	4	40	240
K_new_io	0	6	5	50	300
K_free_io	0	5	4	40	240
K_new_user	0	6	5	50	300
K_free_user	0	5	4	40	240

Table 7: Kernel Memory Management Functions

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
KMB_link	7	141	117	1380	8280
K_link_in	4	78	65	770	4620
K_link_out	4	62	51	630	3780
K_link_assign	4	103	86	980	5880
K_link_control	5	76	63	780	4680
kl_enter	0	27	22	220	1320

Table 8: Kernel Link Management Functions.

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
KSI_bus	1	26	22	250	1500
KHI_bus	0	6	5	50	300
kbus_in	3	49	41	500	3000
kbus_out	0	14	12	120	720
kbus_assign	6	109	91	1090	6540
kbus_control	0	13	11	110	660
KSI_device	1	26	22	250	1500
KHI_device	0	6	5	50	300
kdevice_in	3	48	40	490	2940
kdevice_out	0	10	8	80	480
kdevice_assign	6	109	91	1090	6540
kdevice_control	0	13	11	110	660
KSI_process	1	29	24	270	1620
KHI_process	0	6	5	50	300
kprocess_in	4	56	47	590	3540
kprocess_out	2	40	33	390	2340
kprocess_assign	6	120	100	1180	7080
kprocess_control	0	13	11	110	660

Table 9: Kernel Link Server Functions

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
XB_boot	*	*	*	*	*
XCB_control	*	*	*	*	*
xcontrol_in	*	*	*	*	*
xcontrol_out	*	*	*	*	*
X_command	*	*	*	*	*
list_status	*	*	*	*	*
XC_next_query	1	21	17	200	1200
XC_query_dsm	1	32	27	300	1800
X_report_error	*	*	*	*	*

Note: The code for the functions (for which no data given in this table) is very convoluted. The total number of lines of code in these functions can only be determined at execution time.

Table 10: Executive Controller Functions.

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
X_set_memory	2	20	17	230	1380
X_more_memory	0	6	5	50	300
X_mem_error	0	6	5	50	300
X_allocate	0	9	8	80	480
X_reallocate	0	8	7	70	420
X_free	0	5	4	40	240
X_set_stack	0	9	8	80	480
X_set_queues	0	8	7	70	420
X_add_queue	0	6	5	50	300
X_next_queue	0	7	6	60	360
X_join_queue	0	7	6	60	360
X_find_signature	0	26	22	220	1320
X_set_messages	0	8	7	70	420
X_new_message	0	9	8	80	480
X_free_message	0	5	4	40	240
X_copy_message	1	17	14	170	1020
X_new_buffer	0	7	6	60	360
X_free_buffer	0	9	8	80	480
X_copy_buffer	0	6	5	50	300
X_free_network	0	5	4	40	240
X_free_io	0	5	4	40	240
X_free_user	0	5	4	40	240

Table 11: Executive Utility Functions

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
XCB_router	4	75	62	740	4440
XR_in	*	*	*	*	*
XR_out	*	*	*	*	*
XR_assign	*	*	*	*	*
XR_control	*	*	*	*	*
new_name	1	40	33	360	2160
new_consumer	1	34	28	310	1860
X_route_junction	*	*	*	*	*
X_route_consumer	*	*	*	*	*
XRI_simple	0	9	8	80	480
xsimple_in	*	*	*	*	*
xsimple_out	*	*	*	*	*
xsimple_assign	1	18	15	180	1080
xsimple_control	*	*	*	*	*
XRI_nmr	2	31	26	320	1920
xnmr_in	*	*	*	*	*
xnmr_out	*	*	*	*	*
xnmr_assign	0	11	9	90	540
xnmr_control	*	*	*	*	*
xnmr_out_new	*	*	*	*	*
xnmr_out_error	*	*	*	*	*
xnmr_out_valid	1	23	19	220	1320
xnmr_ready	*	*	*	*	*
xnmr_done	*	*	*	*	*

Note: The code for the functions (for which no data given in this table) is very convoluted. The total number of lines of code in these functions can only be determined at execution time.

Table 12: Executive Routing Manager Functions

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
XCB_manager	*	*	*	*	*
XM_in	*	*	*	*	*
XM_out	*	*	*	*	*
XM_assign	*	*	*	*	*
XM_control	*	*	*	*	*
xm_enter	1	36	30	330	1980
XMI_network	*	*	*	*	*
xnetwork_in	*	*	*	*	*
xnetwork_out	6	93	77	950	5700
xnetwork_assign	*	*	*	*	*
xnetwork_control	*	*	*	*	*
XMI_io	1	45	37	400	2400
xio_in	*	*	*	*	*
xio_out	*	*	*	*	*
xio_assign	*	*	*	*	*
xio_control	*	*	*	*	*
xio_send	6	90	75	930	5580
xio_query	*	*	*	*	*
xi_reset_junction	1	19	16	190	1140

Note: The code for the functions (for which no data given in this table) is very convoluted. The total number of lines of code in these functions can only be determined at execution time.

Table 13(a): Executive Resource Manager Functions

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
XMI_process	1	50	41	440	2640
xprocess_in	*	*	*	*	*
xprocess_out	*	*	*	*	*
xprocess_assign	*	*	*	*	*
xprocess_control	*	*	*	*	*
process_accept	7	100	83	1040	6240
process_query	*	*	*	*	*
process_reply	*	*	*	*	*
process_call	*	*	*	*	*
process_receive	20	224	186	2460	14,760
process_send	*	*	*	*	*
process_ready	16	206	171	2190	13,140
match_query	*	*	*	*	*
process_out_query	12	145	120	1560	9360
process_out_reply	13	179	149	1880	11,280
process_out_send	12	147	122	1580	9480
process_run2wait	8	107	89	1130	6780
process_wait2ready	9	114	95	1220	7320
next_process	7	100	83	1040	6240
xp_reset_junction	1	20	17	200	1200

Note: The code for the functions (for which no data given in this table) is very convoluted. The total number of lines of code in these functions can only be determined at execution time.

Table 13(b): Executive Resource Manager Functions.

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
mc_boot	23	396	329	3980	23,880
mc_add_manager	1	13	11	140	840
mc_add_router	1	16	13	160	960
mc_add_resource	1	35	29	320	1920
mc_add_exec	1	29	24	270	1620
mc_add_link	3	59	49	580	3480
network_link	0	20	17	170	1020
io_link	0	13	11	110	660
appl_link	0	13	11	110	660
mgr_command	*	*	*	*	*
mc_list	*	*	*	*	*
mc_exec_error	0	8	7	70	420
mc_undefined	0	6	5	50	300

Note: The code for the functions (for which no data given in this table) is very convoluted. The total number of lines of code in these functions can only be determined at execution time.

Table 14: DSM Controller Functions.

Module	No. of called Functions	Lines of "C" code (Total)	Lines of "C" code (worst case)	Lines of "assembly" code (worst case)	No. of clock cycles
mc_define	11	315	261	2940	17,640
assign_resource	4	163	135	1470	8820
assign_io	1	69	57	600	3600
assign_appl	1	69	57	600	3600
assign_network	1	104	86	890	5340
sort_execs	0	29	24	240	1440
sort_network	0	34	28	280	1680
mc_link	5	117	97	1120	6720
mc_run	7	75	62	830	4980
run_tos	3	59	49	580	3480
mc_get_consumer	3	64	53	620	3720
mc_get_cpu	6	33	27	450	2700
mc_unknown_con	4	146	121	1330	7980

Table 15: DSM Scheduler Functions.