RA 139 (#41127) 4/12/82
Computer Science 184 pages

THE RESEARCH QUEUEING PACKAGE VERSION 2 CMS USERS GUIDE

Charles H. Sauer, Edward A. MacNair and James F. Kurose

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

Abstract: Queueing networks are important as performance models of systems where performance is principally affected by contention for resources. Such systems include computer systems, communication networks, office systems and manufacturing lines. In order to effectively use queueing networks as performance models, appropriate software is necessary for definition of the networks to be solved, for solution of the networks (by simulation and/or numerical methods) and for examination of the performance measures obtained.

The Research Queueing Package, Version 2 (RESQ) is a system for constructing and solving extended queueing network models. We refer to the class of RESQ networks as "extended" because of characteristics which allow effective representation of system detail. RESQ incorporates a high level language to concisely describe the structure of the model and to specify constraints on the solution. A main feature of the language is the capability to describe models in a hierarchical fashion, allowing an analyst to define submodels to be used analogously to use of macros in programming languages. RESQ also provides a variety of methods for estimating accuracy of simulation results and determining simulation run lengths.

Acknowledgement: We are grateful to P. Heidelberger, E. Jaffe, P. Rosenfeld, M. Reiser, S. Salza, S. Tucci and P.D. Welch for their contributions to RESQ.

This document is the primary documentation for RESQ usage under CMS. A corresponding document exists for TSO usage. Corrections, comments, criticisms and suggestions for improvement of these documents and/or RESQ will be welcomed.

PREFACE

Queueing networks are useful as performance models of systems where performance is principally affected by contention for resources. Such systems include computer systems, communication networks, office systems and manufacturing lines. The Research Queueing Package, Version 2 (hereafter referred to as RESQ) is a system for constructing queueing network models and solving queueing network models. Simulation methods, including state of the art statistical analysis, are provided for the full class of queueing networks allowed in the RESQ language. Numerical methods are provided for a subset of the queueing networks allowed by the RESQ language.

This document describes usage of RESQ with the CMS component of VM/370 and VM/SP. A similar document describes usage of RESQ with the TSO component of OS/VS2 MVS. Though this document is intended to be self contained as far as RESQ usage is concerned, for full effectiveness the user should be familiar with

IBM Virtual Machine/System Product: Introduction, GC19-6200.

IBM Virtual Machine/System Product: CP Command Reference for General Users, SC19-6211.

IBM Virtual Machine/System Product: CMS Command and Macro Reference, SC19-6209.

IBM Virtual Machine/System Product: System Product Editor Command and Macro Reference, SC24-5221.

or corresponding publications. General discussion of performance modeling is given in

C.H. Sauer and K.M. Chandy, *Computer Systems Performance Modeling,* Prentice-Hall, Englewood Cliffs, NJ (1981).

More introductory material on RESQ, examples of networks constructed and solved using RESQ, and discussion of other related publications are given in

C.H. Sauer, E.A. MacNair and J.F. Kurose, "The Research Queueing Package Version 2: Introduction and Examples," IBM Research Report RA-138, Yorktown Heights, New York (April 1982).

This document has the following sections:

"Section 1: Introduction" introduces many of the features and capabilities of RESQ and gives an example of RESQ usage.

"Section 2: The SETUP Command" discusses the command which invokes the RESQ prompter/translator. The RESQ prompter/translator can be used in either interactive or batch ("dialogue file") mode or mixed interactive/batch mode.

Sections 3 through 10 discuss RESQ queueing network elements and corresponding portions of the dialogue language of the RESQ prompter/translator.

"Section 3: Parameters, Identifiers, Variables and Arrays" discusses the dialogue language for declarations of these elements.

April 3, 1982

"Section 4: Active Queues" discusses queueing for resources with timed usage.

"Section 5: Passive Queues" discusses queueing for resources with usage governed by explicit mechanisms for acquiring and freeing units of a resource. Passive queues are some of the most flexible and useful elements in the RESQ language.

"Section 6: Queue Types" discusses a macro facility for queue definition.

"Section 7: Set Nodes" discusses the RESQ elements used to perform assignment statements in the programming language sense.

"Section 8: Split, Fission, Fusion and Dummy Nodes" describes nodes used by jobs for generating other jobs, for synchronizing activities with these jobs and for associated routing definition.

"Section 9: Routing Chains" discusses the definition of routing between network elements, including sources and sinks for jobs and routing decision mechanisms.

"Section 10: Submodels" discusses facilities for macro definition of subnetworks and the invocation of these subnetworks.

"Section 11: Numerical Solution" discusses the restrictions for numerical solution.

"Section 12: Simulation Dialogues" discusses additional language conventions for gathering of distributions, for confidence interval estimation, for run length control and for simulation trace.

"Section 13: The EVAL and EVALT Commands" discusses the two CMS commands available for network solution.

"Section 14: PL/I Embedding" discusses access to RESQ from PL/I procedures as an alternative to use of the EVAL and EVALT commands.

"Appendix 1: Additional Examples" illustrates other aspects of RESQ usage.

"Appendix 2: Names and Keywords" describes the requirements for names of RESQ elements and discusses reserved keywords and names with special meanings.

"Appendix 3: Expressions" describes the rules for expressions used to represent numbers and distributions, including use of user-defined PL/I functions.

"Appendix 4: BNF Grammar" gives a formal definition of the syntax of the dialogue language.

"Appendix 5: SETUP Error Messages" discusses the error messages produced by the prompter/translator.

"Appendix 6: EVAL Error Messages" discusses the error messages produced during network solution.

"Appendix 7: Event Handling" discusses simulation event handling with emphasis on handling of simultaneous events.

"Appendix 8: Installation" discusses installation of RESQ files.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

In many systems, e.g., computing systems, communication networks, automated offices and manufacturing lines, contention for resources (queueing) is a dominant factor in system performance. The interaction between resources and other system elements is often so complex that intuition is insufficient for estimating system performance. Models are used to estimate the performance of systems when measurement of system performance is impossible (e.g., because the system is not yet operational) or impractical (e.g., because of the human and other resources required). Models based on queueing networks are especially useful because such models focus attention on contention for resources.

The basic problems in using queueing network models are to (1) determine the resources and their characteristics which will most affect performance, (2) formulate a model representing these resources and characteristics and (3) determine (by simulation or numerical methods) values for performance measures (e.g, mean response time) in the model. The first two of these problems are highly system specific. Thus we will not address these problems directly. The Research Queueing Package (RESQ) is a software *tool* for building queueing network models. We emphasize "tool" because RESQ is not a model itself but rather a facility for constructing and developing a model. As a tool, it can be of great value in dealing with the second and third basic problems cited above.

In the following sections (1.1 - 1.3) we present a brief overview of RESQ, and as an example, use RESQ to develop a queueing network model of an interactive computing system. This example is intended to illustrate many of the facilities of RESQ. Three additional examples are given in Appendix 1: (1) a very simple model solved numerically, (2) a model which further develops the example in Section 1.3, and (3) a model of a simple communication network used for access to an interactive computer system.

1.1. RESQ Diagrams

Effective use of RESQ is based on constructing diagrams representing queueing network models. Figure 1.1 illustrates a simple queueing network model of an interactive computer system. (This network is similar to networks used as computer system models since the mid sixties.)

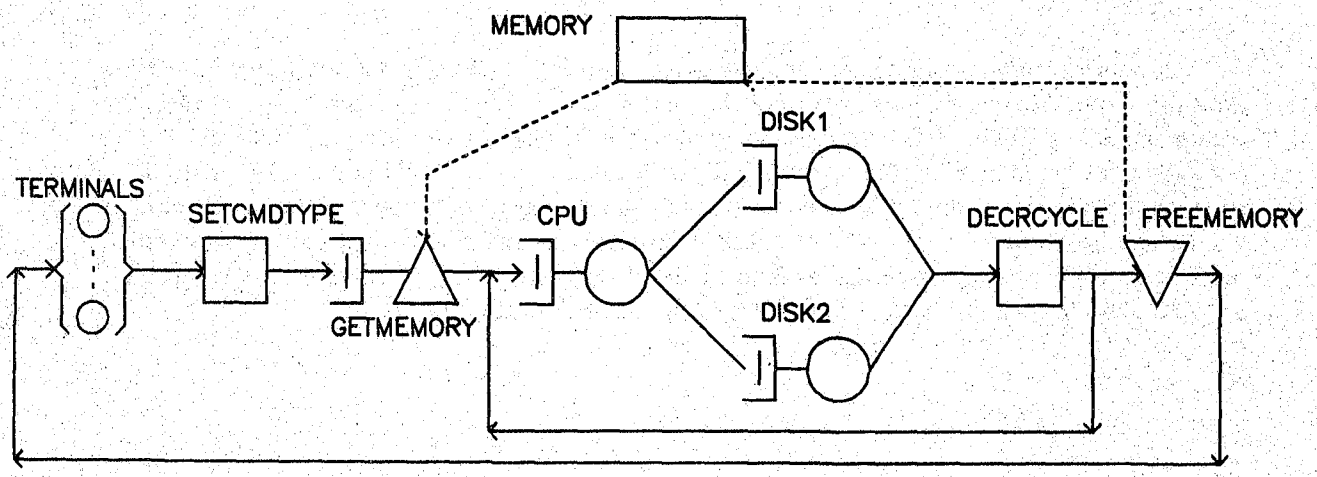

Figure 1.1 - Queueing Network Model

The symbols in Figure 1.1 represent specific elements in the RESQ diagram language. Figures 1.2 - 1.4 show the symbols for all such elements. Descriptions of the RESQ symbols will be given in later sections which discuss the corresponding RESQ elements. The model considers contention for three kinds of system resources, main memory, a CPU and disk memory, and represents the terminals as well. Users of the system are represented by *jobs* in the queueing network. Part of a user's time is spent thinking at the terminal and keying in commands; this part of the user's time is represented by service times of a job (representing the user) at the terminals queue. The model assumes there are as many terminals as users, so there is no waiting for a terminal; we will still refer to the model representation of the terminals as an "infinite server queue." After thinking and keying in a command, the user spends the remaining part of his or her time (for this interaction) waiting for a response. The job representing the user waits to receive main memory. Once it receives main memory, this job alternates between computation and I/O activities until the command processing is finished, main memory is released and the user receives the response. The user then begins another thinking/keying time.

In using RESQ to model systems, the most difficult steps are usually those of describing system resources and activities as we have just done and developing a corresponding diagram, e.g., Figure 1.1. Also, one must obtain data for amounts of resources required, times spent holding resources, frequency of resource requests, etc. Having the description, diagram and data, construction and solution of the model using RESQ is an efficient and straightforward process.

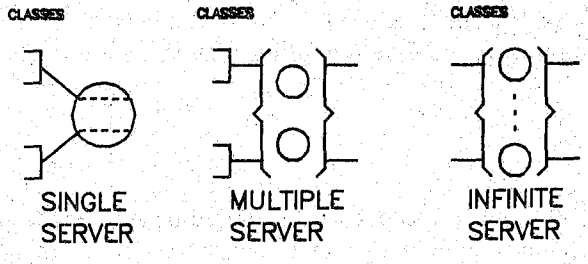

Figure 1.2 - Active Queues
Single, Multiple, Infinite Server

As in programming, in system modeling it is helpful to develop hierarchical representations of models in order to clarify models, permit the refinement of models and ease the maintenance of models. RESQ provides a macro-like facility for developing "submodels," i.e., parameterized templates of subnetworks. In the example of Figure 1.1 it would be natural to have a submodel consisting of the queues of the computer system (excluding the terminals). It would also be natural to represent the disk subsystems as submodels in case a more detailed representation of the disk subsystems is to be developed later. Figure 1.5 depicts the top level of such a hierarchy and Figure 1.6 depicts the middle and bottom levels.

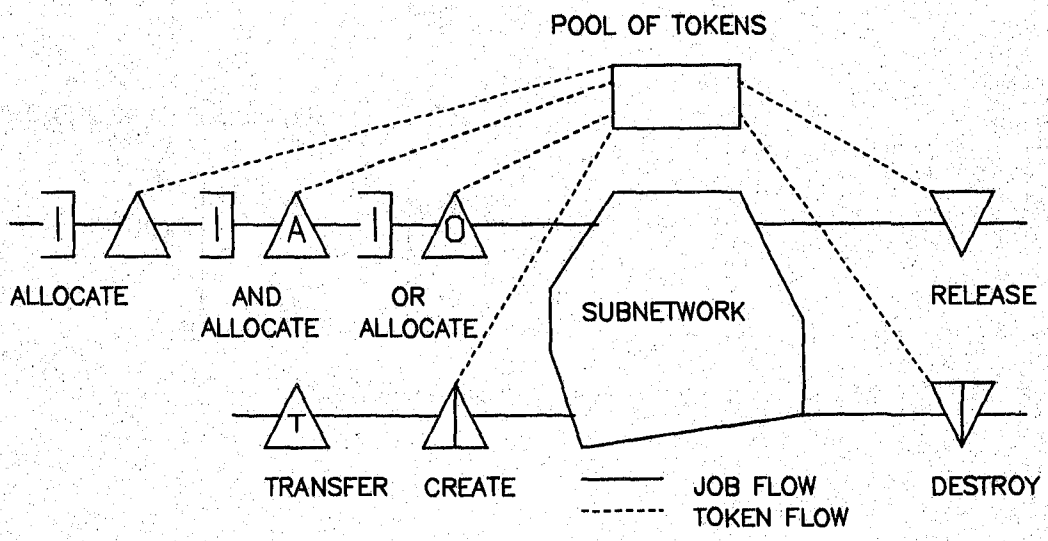

Figure 1.3 - Passive Queue

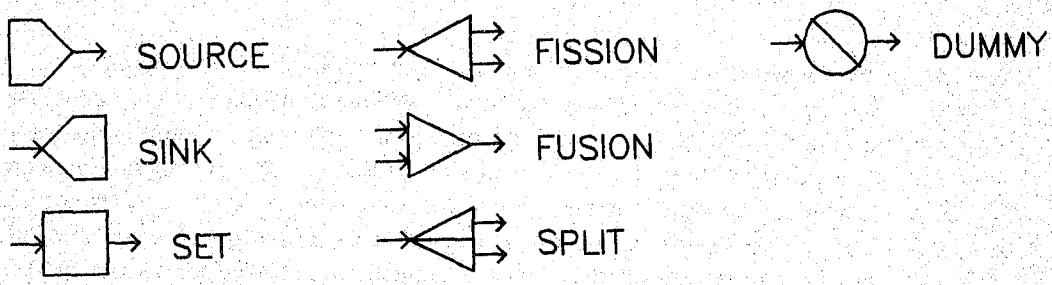

Figure 1.4 - Symbols for Other Nodes

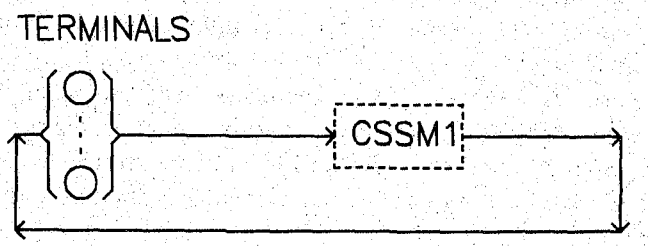

Figure 1.5 - Terminals and Submodel

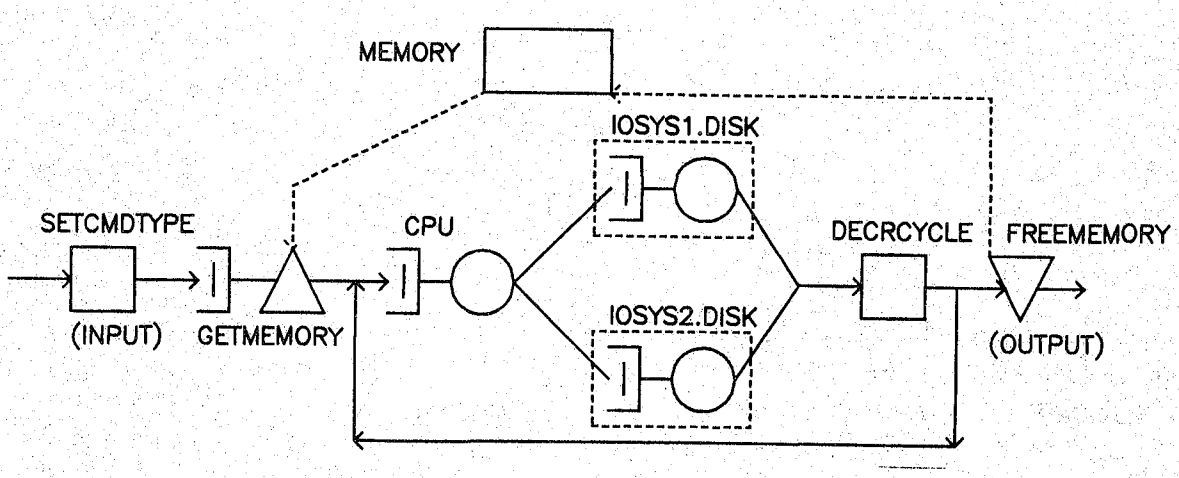

Figure 1.6 - Computer System Submodel

1.2. RESQ Elements

In this section we briefly describe some of the elements of RESQ queueing networks that apply to the above example. We refer to the networks of RESQ as "extended" because of characteristics absent from classical queueing models. Classical queues are "active" queues in RESQ terminology. A job's activity is typically focused on the resources of active queues. A job typically has no interaction with other model elements while at an active queue.

Perhaps the most important of the extensions introduced in extended networks is the "passive" queue, which allows convenient representation of simultaneous resource possession. A job typically acquires units of a passive queue resource and holds on to them while visiting other queues (including other passive queues) and model elements. The job explicitly releases the units of resource when it no longer needs them. In our computer system example, a job must hold memory while using the processor or I/O devices; a passive queue may be used to represent this holding of memory. Additional passive queues could be added to the model to represent contention for channels, device controllers, etc.

As well as representing simultaneous resource possession, passive queues often allow simple representations of complex mechanisms. For example, in a system where a channel is shared among position sensing I/O devices and the channel is not held during positioning, contention for the channel may cause jobs to wait for extra revolutions after positioning before the channel is acquired and data transfer takes place. This situation can be modeled by use of a passive queue representing the channel and a status function testing availability of the channel, as illustrated in Appendix 1. Communication network protocols and similar mechanisms are often conveniently modeled by passive queues, as also illustrated in Appendix 1.

A third use of passive queues is in measuring response times in subnetworks. The "queueing time" (response time) for a passive queue is defined as the time between a job's request for units of the passive queue resource and that job's freeing of the units of resource. Thus in our example the queueing time for the passive queue corresponds to the response time seen by the terminal users.

1.3. RESQ User Interfaces

The RESQ user interfaces have been designed for effective use by both novice and advanced users. The user interfaces are based on interactive dialogues which serve to educate new users working with small models. The interactive dialogues provide optional tutorials to clarify prompts. Transcripts of interactive dialogue can be easily used to revise and develop models. There are two basic sets of dialogue, a model definition dialogue and a model solution dialogue.

The SETUP command invokes the RESQ prompter/translator for definition or revision of a model. The prompter automatically provides for immediate correction of syntactic errors. If a RESQ user realizes a semantic error was made in some previous portion of the dialogue, he or she may temporarily suspend the dialogue, correct the error and then resume the dialogue at the point of suspension. A transcript (a "dialogue file") of a model definition dialogue is kept for the user. The user may edit this transcript and then have it translated again, with or without additional interactive dialogue. The EVAL command is used to solve (e.g., to simulate) a model.

We will now give an example of a possible SETUP dialogue for the model represented by Figures 1.5 and 1.6. As we present the dialogue we will make arbitrary assumptions about system characteristics previously left unspecified. The example is presented as if a typewriter-type terminal is used, to simplify formatting of this document. However, RESQ is insensitive to the type of terminal used and is typically used with a display terminal.

In our examples, upper case characters will correspond to prompts from RESQ components and lower case will generally be used for replies from the user. Prompts are always terminated by a colon (":"). RESQ generates some additional heading lines for sections of dialogue; these heading lines do not require replies from the user. RESQ is insensitive to upper/lower case, but preserves case in listing and transcript files.

The following example will be interspersed with discussion explaining the portions of dialogue. A contiguous transcript follows the example dialogue. Assuming we are in the CMS environment with access to the mini-disk containing the RESQ files and with sufficient virtual storage, we issue the SETUP command, are prompted for a model name and, after asking for a tutorial with the special reply "how", give the name "csm" for "computer system model."

```
setup
MODEL:how
      MODEL NAME MUST START WITH A LETTER,
      CONSIST OF ONLY LETTERS AND DIGITS
      AND HAVE AT MOST EIGHT CHARACTERS
MODEL:csm
RESQ2 Translator V2.04 (01/19/82)  Time: 13:56:12  Date: 01/29/82
MODEL IS CSM
```

Except for model names, which are constrained to fewer characters for compatibility with CMS and TSO, names of RESQ elements may be up to ten characters long. Next we indicate the solution method, either simulation or numerical:

```
METHOD:simulation
```

The first major section of dialogue is used to declare parameters which will be defined when the solution is performed, to declare identifiers representing expressions and to declare the extent of JV, the vector of variables associated with each job. Solution may be performed

repeatedly for different parameter values without reissuing the SETUP or EVAL commands. We may list as many parameters as we wish -- SETUP will continue to prompt for parameters until we give a null reply. If more than one name is listed on the same line, then the names are separated by either blanks or commas (","). Our examples will usually use blanks rather than commas. RESQ treats multiple blanks as equivalent to a single blank.

```
   NUMERIC PARAMETERS:thinktime users
   NUMERIC PARAMETERS:
```

Identifiers are provided to allow naming of expressions (typically, but not necessarily, numeric constants) for sake of clarity and to allow changes to be made without searching for all instances of an expression. SETUP expects a list of identifier names. For each name, SETUP will prompt for an expression for the value associated with the identifier name. SETUP will prompt for additional lists of names until given a null reply.

```
   NUMERIC IDENTIFIERS:userframes
      USERFRAMES:50
   NUMERIC IDENTIFIERS:
```

In our example we assume that there are three types of commands which may be issued by terminal users. JV(0) will be used to store the command type, and JV(1) will be used to count the number of CPU-I/O cycles for a particular command. We include a comment to indicate this usage. Comments may be included in replies using the PL/I convention, i.e., a comment is a string beginning with "/*", ending with "*/" and not otherwise containing "*/". A comment must end on the same line it begins on. (As discussed in Section 2.2, multiple physical lines may be concatenated to give the effect of a single logical line.)

```
   MAX JV:how
ENTER AN ARITHMETIC EXPRESSION FOR THE EXTENT OF THE JV VECTOR
   MAX JV:1 /*0: command type, 1: cycle count*/
```

The second major section of dialogue is for definition of queues. First we may define a "queue type", a macro definition of a queue dialogue. We indicate here that we choose not to define a queue type by giving a null reply. We will illustrate definition and invocation of a user defined queue type later in the dialogue.

```
   QUEUE TYPE:
```

Next we define individual queues. The only queue outside of the submodel in the network of Figure 1.5 is the terminals queue. This queue is assumed to have at least as many servers as jobs in the network, i.e., it is an "Infinite Server" (IS) queue. We use the predefined IS type, which indicates an active queue with default characteristics, rather than the general ACTIVE type. First we are prompted for the queue name, then the queue type.

```
   QUEUE:terminalsq
      TYPE:how
VALID QUEUE TYPES ARE: ACTIVE, FCFS, IS, LCFS, PRTY, PRTYPR, PS,
                       PASSIVE OR A USER DEFINED TYPE
      TYPE:is
```

In addition to its servers, an active queue has one or more waiting lines called "classes." Routing definitions will use the class names, not the queue name. The active queues in our example each have only one class. After prompting for a list of classes, SETUP will prompt for the service time distributions associated with the classes. In the following we give the

name of a numeric parameter, which will be interpreted as the mean of an exponential distribution. SETUP will prompt for more classes until a null reply is given. SETUP will then prompt for more queue definitions until a null reply is given.

```
      CLASS LIST:terminals
         SERVICE TIMES:thinktime
      CLASS LIST:
   QUEUE:
```

The third major section of dialogue is for definition of additional nodes not belonging to queues. "Nodes" in RESQ are functional elements in the routing, including classes, the elements shown in Figure 1.3 except for the pool of tokens and all elements shown in Figure 1.4. None of these nodes appear outside of the submodel of Figure 1.5, so we give null replies to the prompts for names of these nodes.

```
SET NODES:
FISSION NODES:
FUSION NODES:
```

The fourth major section of dialogue is for definition of submodels. The submodel definition dialogue closely parallels the dialogue for model definition, including subsections corresponding to those sections we have already seen and a routing subsection corresponding to the model routing section which follows submodel definition and invocation. First we are prompted for a name of the submodel definition. Then we are prompted for parameter and identifier definitions.

```
SUBMODEL:cssm /*Computer System Submodel*/
   NUMERIC PARAMETERS:pageframes
   NUMERIC PARAMETERS:
```

Node parameters provide for reference to nodes outside the submodel from within the submodel. We do not need node parameters with this example.

```
   NODE PARAMETERS:
```

Routing "chains" are used to define the routing among nodes of a network. A submodel must have at least one chain parameter in order to connect the nodes inside of the subnetwork with nodes outside of the subnetwork.

```
   CHAIN PARAMETERS:interactiv
   CHAIN PARAMETERS:
   NUMERIC IDENTIFIERS:cmdtype cyclecount
      CMDTYPE:0 /*JV(0) to be used to indicate command type*/
      CYCLECOUNT:1 /*JV(1) to be used to count CPU-I/O cycles*/
```

Numeric parameters and identifiers may be defined as one or two-dimensional arrays. Unlike the special case of JV, which allows zero as an index, the indices of these arrays begin at one. In our example we have three command types with different numbers of CPU-I/O cycles associated with each type and with different requirements for page frames for each type.

```
   NUMERIC IDENTIFIERS:cpiocycles(3) pageneed(3)
      CPIOCYCLES: 8 15 50
      PAGENEED:  20 24 30
   NUMERIC IDENTIFIERS:cputime
```

```
      CPUTIME:.025 /*mean time in seconds*/
NUMERIC IDENTIFIERS:
QUEUE TYPE:
```

A passive queue consists of a pool of tokens to be allocated to jobs and a set of nodes which operate on that pool and the jobs holding tokens. The passive queue in our example has one token for each page frame. After prompts for the queue name and queue type, we are prompted for the number of tokens initially in the pool and the scheduling discipline. We choose "first come first served" (fcfs) scheduling.

```
QUEUE:memory
   TYPE:passive
   TOKENS:pageframes
   DSPL:fcfs
```

Next we are prompted for lists of names of allocate nodes. Jobs wait at allocate nodes until they are given the number of tokens they request and then move without delay to the next node in the routing chain for the allocate node. (In the interactive mode, SETUP gives prompts only for "plain" allocate nodes. Section 6 will discuss the other kinds of allocate nodes shown in Figure 1.3.)

```
ALLOCATE NODE LIST:getmemory
   NUMBERS OF TOKENS TO ALLOCATE:pageneed(jv(cmdtype))
ALLOCATE NODE LIST:
RELEASE NODE LIST:freememory
```

A job visiting a release node returns all tokens it received from a passive queue. This takes no simulated time.

```
RELEASE NODE LIST:
DESTROY NODE LIST:
CREATE NODE LIST:
```

We assume the CPU has a single server with the "processor sharing" (PS) scheduling discipline. PS is the limiting case of a round robin ("time slicing") discipline when the quantum ("time slice") tends to zero, provided there is negligible overhead in switching from job to job. The service times are assumed to have an exponential distribution with mean given by the identifier "cputime."

```
QUEUE:cpuq
   TYPE:ps
   CLASS LIST:cpu
      SERVICE TIMES:cputime
   CLASS LIST:
QUEUE:
```

In our example, when a job leaves the terminals we wish to determine its command type by random selection and also store the number of CPU-I/O cycles for that type in JV(1). We assume that the command is type 1 with probability 0.8, type 2 with probability 0.15 and type 3 with probability 0.05. A reference to the RESQ "discrete" distribution, "discrete(1,.8;2,.15;3,.05)", will be used to make this random selection. The semicolons (";") are used to separate the major pairs of values given to the discrete distribution. Semicolons are used to separate important expressions or lists of expressions in RESQ. The commas separating the pairs of values may be replaced by blanks, as we discussed previously.

Set nodes are used to perform assignment statements in the sense of programming languages. The assignments may be made to job variables or to two other kinds of variables we will introduce in Section 8 of this document. The "SET NODES:" prompt requests a list of names of set nodes. If the list contains more than one name, then the nodes in the list may perform only one assignment. Otherwise, if the list contains only one name, then several assignments may be performed by that one node. In the following we will wish to list the two assignments on a single logical line, but will not have room to do so on a single physical line. SETUP allows use of "++" at the end of a physical line to indicate the next physical line is to be concatenated with the current logical line of input to form a single logical line. SETUP will prompt with a colon (":") for additional physical line(s).

```
SET NODES:setcmdtype
   ASSIGNMENT LIST:jv(cmdtype)=discrete(1,.8;2,.15;3,.05), ++
:jv(cyclecount)=cpiocycles(jv(cmdtype))
```

The assignments are performed in the order listed. In the above definition, the value of JV(CMDTYPE) used in the second assignment will be the value given by the first assighment.

When a job completes a CPU-I/O cycle, we will decrement JV(1) and test to see if the job has completed its count by testing for JV(1)=0.

```
SET NODES:decrcycles
   ASSIGNMENT LIST:jv(cyclecount)=jv(cyclecount)-1
SET NODES:
FISSION NODES:
FUSION NODES:
```

The following submodel definition is very sparse, but could be embellished considerably without changing its subsequent invocations in the submodel cssm.

```
SUBMODEL:iosys
   NUMERIC PARAMETERS:
   NODE PARAMETERS:
   CHAIN PARAMETERS:interactiv
   CHAIN PARAMETERS:
   NUMERIC IDENTIFIERS:
```

In this example we reuse the name "interactiv" for the chain parameter. As in nested procedure definitions in block structured programming languages (e.g., PL/I or Pascal), names used outside of a submodel definition may be reused within submodel definitions. When names are reused in this manner, the new definition persists within the submodel definition and the old definition is restored after the submodel definition is completed. On the other hand, we could have used an entirely different name for the chain parameter in this example.

In our example we assume each disk is represented by a single server queue with a single service time representing positioning and transfer and with fcfs scheduling. The RESQ "fcfs" predefined queue type would naturally be used in this instance. However, we will illustrate a user defined queue type with these same assumptions with the added assumptions that service times have an exponential distribution with mean 0.06. First we are prompted for the name of the queue type and names of any numeric parameters.

```
QUEUE TYPE:diskdef
   NUMERIC PARAMETERS:
```

Every node, e.g., every class, to be associated with a queue defined by a user defined queue type must be declared as a node parameter of the queue type. In this case we will have only one class.

```
NODE PARAMETERS:servicecls
NODE PARAMETERS:
```

After declaring the parameters, the rest of a queue type definition is very similar to a queue definition. The reply to "TYPE:" may be any predefined queue type. Because the general active type allows defining individual servers, the prompt for "SERVICE TIMES:" which we have seen previously is replaced by "WORK DEMANDS:". The work demanded of a server is divided by the service rate to get service times. All of our previous dialogues have assumed a unit rate server, so work demands and service times are equivalent in these dialogues. In the following we will avoid defining a server, so we will get a unit rate server by default.

```
   TYPE:active
   SERVERS:1
   DSPL:fcfs
   CLASS LIST:servicecls
      WORK DEMANDS:.06
   CLASS LIST:
   SERVER-
      RATES:
END OF QUEUE TYPE DISKDEF
QUEUE TYPE:
```

A definition of a queue defined by a user defined queue type, i.e., an invocation of a queue type definition, will consist of prompts for parameter values after the queue type is specified. In this case there is only one parameter, the class.

```
QUEUE:diskq
   TYPE:diskdef
   SERVICECLS:disk
QUEUE:
SET NODES:
FISSION NODES:
FUSION NODES:
```

The following prompts give us the opportunity to define submodels within this submodel (iosys) definition and to invoke submodel definitions within the definition of iosys. Since we have not finished defining iosys, we are not ready to invoke iosys.

```
SUBMODEL:
INVOCATION:
```

We have not seen any routing chain definitions yet. The following definition is atypical because within the submodel there is only one node, "disk", and so no routing within the submodel will be defined. After giving the name of the chain, we indicate that this chain is to be completed in the external model, i.e., in the model invoking the submodel "iosys". We then indicate that "disk" is both the standard entry point and the standard exit point of the chain. In the invoking model we will refer to "disk" by the synonyms "input" and "output". The colon prompt (":") is for a routing transition, as we will see below.

```
   CHAIN:interactiv
      TYPE:external
      INPUT:disk
      OUTPUT:disk
      :
   CHAIN:
END OF SUBMODEL IOSYS
SUBMODEL:
```

An invocation of a submodel is an instance of the subnetwork defined by the submodel definition and the parameters specified with the invocation. After a prompt for the invocation name, there is a prompt for the name of the submodel definition to be used in this invocation. Assuming only the name of the submodel definition is given, there will be additional prompts for the parameter values, in this case the name of the chain parameter.

```
INVOCATION:iosys1
   TYPE:iosys
   INTERACTIV:interactiv
```

The prompt "INTERACTIV:" requests the value for the parameter name defined within the submodel IOSYS definition. The reply "interactiv" supplies a value, the chain parameter declared within the submodel CSSM definition. These names happen to be the same in our example, but there is no requirement that they be the same.

For brevity, the prompts for parameter values can be avoided by supplying the parameter values (in the order the parameters were declared) after the name of the submodel definition, e.g.,

```
INVOCATION:iosys2
   TYPE:iosys: interactiv
INVOCATION:
```

This invocation creates a second subnetwork with the characteristics of submodel IOSYS. (In this case the subnetwork consists only of a single queue.)

The following chain definition is more typical than the previous one. After declaring the standard entry point to be the set node setcmdtype and the standard exit point to be the release node freememory, we define the routing among the nodes of the subnetwork.

```
CHAIN:interactiv
   TYPE:external
   INPUT:setcmdtype
   OUTPUT:freememory
```

The following line indicates that jobs leaving setcmdtype always go to the allocate node getmemory, that jobs leaving getmemory always go to the class cpu and that jobs leaving cpu go to the standard entry of invocation iosys1 (disk) with probability 0.5 and to the standard entry of invocation iosys2, also with probability 0.5.

```
:setcmdtype->getmemory->cpu->iosys1.input iosys2.input;.5 .5
```

The following line indicates that jobs leaving the standard exit of either iosys1 or iosys2 go to set node decrcycles.

```
        :iosys1.output iosys2.output->decrcycles
```

The following line indicates that jobs leaving decrcycles return to cpu if JV(1) is positive and go to the release node otherwise. The "t" in "if(t)" represents "true".

```
        :decrcycles->cpu freememory;if(jv(cyclecount)>0) if(t)
        :
    CHAIN:
END OF SUBMODEL CSSM
SUBMODEL:
```

Following is the invocation of the submodel representing the entire computer system, with values for the numeric and chain parameters.

```
INVOCATION:cssm1
    TYPE:cssm
    PAGEFRAMES:userframes
    INTERACTIV:interactiv
INVOCATION:
```

A chain in the model proper will be either open, if there are to be provisions for external arrivals and departures, or closed, if jobs are fixed within the chain (as in our example). With a closed chain we must indicate the population, i.e., the number of jobs fixed within the chain.

```
CHAIN:interactiv
    TYPE:closed
    POPULATION:users
    :terminals->cssm1.input
    :cssm1.output->terminals
    :
CHAIN:
```

This completes definition of the model proper. The remaining dialogue section pertains to the specifics of simulation solution.

Many performance measures are gathered by the simulation by default. However, gathering of distributions of these measures for all appropriate network elements can be expensive in both time and memory, so distributions are only gathered when requested. In our example model the most interesting distribution is likely to be the distribution of response times seen by the terminal users. The queueing time for the passive queue, defined as the time of arrival at the allocate node to departure from the release node, will be this desired response time. The following requests that the cumulative queueing time distribution be gathered for queueing times from 1 to 8 at unit intervals. The name of the memory passive queue must be qualified by the invocation name, "cssm1," when it is referred to outside of the submodel definition.

```
QUEUES FOR QUEUEING TIME DIST:cssm1.memory
    VALUES:1 2 3 4 5 6 7 8
QUEUES FOR QUEUEING TIME DIST:
```

We also request that the queue length distribution for the passive queue be gathered for all possible lengths. Just as queueing time includes time holding tokens, queue length includes jobs holding tokens.

```
   QUEUES FOR QUEUE LENGTH DIST:cssm1.memory
   MAX VALUE:users
   QUEUES FOR QUEUE LENGTH DIST:
   NODES FOR QUEUEING TIME DIST:
   NODES FOR QUEUE LENGTH DIST:
```

RESQ provides three methods for estimating confidence intervals for performance measures, and two of these three methods also provide for run length control based on the confidence intervals. In this example we will not illustrate confidence interval estimation or associated run length control.

```
   CONFIDENCE INTERVAL METHOD:how
CONFIDENCE INTERVAL METHODS ARE: REGENERATIVE, REPLICATIONS, SPECTRAL
                                 OR NONE
   CONFIDENCE INTERVAL METHOD:none
```

For closed routing chains (and open chains which are not initially empty) we must specify where the jobs of the chain are to be placed initially.

```
   INITIAL STATE DEFINITION-
   CHAIN:interactiv
      NODE LIST:terminals
      INIT POP:users
   CHAIN:
```

The simulation run will end when the first of the following limits are reached. Simulated events in this model will correspond exactly to the completions of service at the active queues. Departures from the passive queue will correspond exactly to the visits to the release node.

```
   RUN LIMITS-
      SIMULATED TIME:3600
      EVENTS:50000
      QUEUES FOR DEPARTURE COUNTS:cssm1.memory
         DEPARTURES:500
      QUEUES FOR DEPARTURE COUNTS:
      NODES FOR DEPARTURE COUNTS:
   LIMIT - CP SECONDS:5
```

Specification that there will be no simulation trace ends definition of this model.

```
   TRACE:how
ENTER EITHER 'YES' OR 'NO'
   TRACE:no
END
 NO FATAL ERRORS DETECTED DURING COMPILATION.
```

Once we have completed the SETUP dialogue, a transcript of the dialogue (a "dialogue file") is available in a file on mini-disk A with file name the same as the model name and file type RQ2INP. This transcript can be edited and used as input to the SETUP command, thus avoiding repeating the dialogue to make minor changes. Use of the dialogue file mode of SETUP provides capabilities for language elements not available in the interactive mode. Most importantly, it is possible to include dialogue fragments, e.g., submodel definitions, from libraries of dialogue. For our example model, file CSM RQ2INP A1 is

```
MODEL:CSM
   METHOD:simulation
   NUMERIC PARAMETERS:thinktime users
   NUMERIC IDENTIFIERS:userframes
      USERFRAMES:50
   MAX JV:1 /*0: command type, 1: cycle count*/
   QUEUE:terminalsq
      TYPE:is
      CLASS LIST:terminals
         SERVICE TIMES:thinktime
   SUBMODEL:cssm /*Computer System Submodel*/
      NUMERIC PARAMETERS:pageframes
      CHAIN PARAMETERS:interactiv
      NUMERIC IDENTIFIERS:cmdtype cyclecount
         CMDTYPE:0 /*JV(0) to be used to indicate command type*/
         CYCLECOUNT:1 /*JV(1) to be used to count CPU-I/O cycles*/
      NUMERIC IDENTIFIERS:cpiocycles(3) pageneed(3)
         CPIOCYCLES: 8 15 50
         PAGENEED:  20 24 30
      NUMERIC IDENTIFIERS:cputime
         CPUTIME:.025 /*mean time in seconds*/
      QUEUE:memory
         TYPE:passive
         TOKENS:pageframes
         DSPL:fcfs
         ALLOCATE NODE LIST:getmemory
            NUMBERS OF TOKENS TO ALLOCATE:pageneed(jv(cmdtype))
         RELEASE NODE LIST:freememory
      QUEUE:cpuq
         TYPE:ps
         CLASS LIST:cpu
            SERVICE TIMES:cputime
      SET NODES:setcmdtype
         ASSIGNMENT LIST:jv(cmdtype)=discrete(1,.8;2,.15;3,.05), ++
jv(cyclecount)=cpiocycles(jv(cmdtype))
      SET NODES:decrcycles
         ASSIGNMENT LIST:jv(cyclecount)=jv(cyclecount)-1
      SUBMODEL:iosys
         CHAIN PARAMETERS:interactiv
         QUEUE TYPE:diskdef
            NODE PARAMETERS:servicecls
            TYPE:active
            SERVERS:1
            DSPL:fcfs
            CLASS LIST:servicecls
               WORK DEMANDS:.06
            SERVER -
         END OF QUEUE TYPE DISKDEF
         QUEUE:diskq
            TYPE:diskdef
            SERVICECLS:disk
         CHAIN:interactiv
            TYPE:external
            INPUT:disk
```

```
            OUTPUT:disk
      END OF SUBMODEL IOSYS
      INVOCATION:iosys1
         TYPE:iosys
         INTERACTIV:interactiv
      INVOCATION:iosys2
         TYPE:iosys: interactiv
      CHAIN:interactiv
         TYPE:external
         INPUT:setcmdtype
         OUTPUT:freememory
         :setcmdtype->getmemory->cpu->iosys1.input iosys2.input;.5 .5
         :iosys1.output iosys2.output->decrcycles
         :decrcycles->cpu freememory;if(jv(cyclecount)>0) if(t)
   END OF SUBMODEL CSSM
   INVOCATION:cssm1
      TYPE:cssm
      PAGEFRAMES:userframes
      INTERACTIV:interactiv
   CHAIN:interactiv
      TYPE:closed
      POPULATION:users
      :terminals->cssm1.input
      :cssm1.output->terminals
   QUEUES FOR QUEUEING TIME DIST:cssm1.memory
      VALUES:1 2 3 4 5 6 7 8
   QUEUES FOR QUEUE LENGTH DIST:cssm1.memory
   MAX VALUE:users
   CONFIDENCE INTERVAL METHOD:none
   INITIAL STATE DEFINITION -
   CHAIN:interactiv
      NODE LIST:terminals
      INIT POP:users
   RUN LIMITS -
      SIMULATED TIME:3600
      EVENTS:50000
      QUEUES FOR DEPARTURE COUNTS:cssm1.memory
         DEPARTURES:500
   LIMIT - CP SECONDS:5
   TRACE:no
END
```

The dialogue file does not include SETUP prompts which received null replies. Note that the indentation provided by SETUP clarifies the structure of the model, particularly the nesting of submodels. (In user creation or modification of dialogue files, the user is free to use other indentation conventions, including no indentation.)

The EVAL command invokes dialogue for model solution (e.g., simulation). This dialogue prompts the user for parameter values, performs the solution and then provides the user with performance measures requested by the user. When this dialogue is complete for a particular set of parameter values, it may be repeated for a different set of parameter values without reissuing the EVAL command.

For our example model, we might have the following dialogue with the EVAL command. First we are prompted for the model name and values for parameters.

```
eval
RESQ2 EXPANSION AND SOLUTION PROGRAM.
MODEL:how
NAME OF MODEL ALREADY DEFINED WITH SETUP
MODEL:csm
RESQ2 VERSION DATE: JANUARY 29, 1982 -  TIME: 17:00:35  DATE: 01/29/82
THINKTIME:16
USERS:25
```

Once the parameter values are specified, the model definition is complete and macro-expansion of the submodel definitions is performed. Then solution commences. When simulation ends, we get one or more messages indicating why simulation stopped, an error message or a message indicating no errors were detected, and a summary of the simulation run.

```
RUN END: CPU LIMIT
NO ERRORS DETECTED DURING SIMULATION.
                  SIMULATED TIME:       245.77480
                        CPU TIME:            5.25
                 NUMBER OF EVENTS:            7461
```

Then we are prompted "WHAT:" meaning "What performance measures do you want to see?". A reply of "all" results in a display of all measures normally provided. Instead of "all", we give the code "nd" for number of departures and specify we are only interested in the passive queue. (A reply of "how" would provide a tutorial listing all such codes.)

```
WHAT:nd(cssm1.memory)
INVOCATION      INVOCATION      ELEMENT         NUMBER OF DEPARTURES
                CSSM1           MEMORY          324
```

We then give the code "qt" for mean queueing time and specify we are only interested in the passive queue. (This queueing time is the response time seen by terminal users in the modeled system.)

```
WHAT:qt(cssm1.memory)
INVOCATION      INVOCATION      ELEMENT         MEAN QUEUEING TIME
                CSSM1           MEMORY          2.81971
```

A null reply to "WHAT:" terminates the examination of performance measures.

```
WHAT:
```

We are then given the opportunity to extend the simulation run. We may increase any of the run limits we specified before and let the simulation run until one of the new limits is reached. In the following we increase the limit on CPU time. (This example was run on a model 3033 processor.)

```
CONTINUE RUN:yes
LIMIT - SIMULATED TIME:how
LARGER VALUE THAN  3.600E+03 OR NULL TO KEEP THAT VALUE
TRY AGAIN-
```

```
LIMIT - SIMULATED TIME:
LIMIT - EVENTS:
LIMIT - CSSM1.MEMORY DEPARTURES:
LIMIT - CP SECONDS:10
```

When the simulation reaches one of the new limits, we see the old termination message followed by a new one and a new summary of the simulation run. We then receive the "WHAT:" prompt again.

```
RUN END: CPU LIMIT
RUN END: CSSM1.MEMORY DEPARTURE LIMIT
NO ERRORS DETECTED DURING SIMULATION.
                  SIMULATED TIME:      366.25098
                        CPU TIME:        8.10
                NUMBER OF EVENTS:       11528

WHAT:nd(cssm1.memory)

INVOCATION      INVOCATION      ELEMENT         NUMBER OF DEPARTURES
                CSSM1           MEMORY          500

WHAT:qt

INVOCATION      INVOCATION      ELEMENT         MEAN QUEUEING TIME
                                TERMINALSQ      14.82022
                CSSM1           MEMORY          2.95095
                CSSM1           CPUQ            0.03118
CSSM1           IOSYS1          DISKQ           0.07692
CSSM1           IOSYS2          DISKQ           0.07245
```

The utilization measure is the fraction of time a server or token is in use.

```
WHAT:ut

INVOCATION      INVOCATION      ELEMENT         UTILIZATION
                                TERMINALSQ      0.00000
                CSSM1           MEMORY          0.72492
                CSSM1           CPUQ            0.37966
CSSM1           IOSYS1          DISKQ           0.45804
CSSM1           IOSYS2          DISKQ           0.42182

WHAT:
```

We choose to extend the run again, this time intending to reach 1000 departures from the memory queue.

```
CONTINUE RUN:yes
LIMIT - SIMULATED TIME:
LIMIT - EVENTS:
LIMIT - CSSM1.MEMORY DEPARTURES:
1000
LIMIT - CP SECONDS:20
RUN END: CPU LIMIT
RUN END: CSSM1.MEMORY DEPARTURE LIMIT
```

```
RUN END: CSSM1.MEMORY DEPARTURE LIMIT
NO ERRORS DETECTED DURING SIMULATION.
                SIMULATED TIME:      753.42139
                      CPU TIME:          16.29
               NUMBER OF EVENTS:          23054

WHAT:qt(cssm1.memory)

INVOCATION      INVOCATION      ELEMENT         MEAN QUEUEING TIME
                CSSM1           MEMORY          2.78478
```

Now we examine all of the normally provided performance measures.

```
WHAT:all

INVOCATION      INVOCATION      ELEMENT         UTILIZATION
                                TERMINALSQ      0.00000
                CSSM1           MEMORY          0.71155
                CSSM1           CPUQ            0.37169
CSSM1           IOSYS1          DISKQ           0.44270
CSSM1           IOSYS2          DISKQ           0.42385

INVOCATION      INVOCATION      ELEMENT         THROUGHPUT
                                TERMINALSQ      1.32861
                CSSM1           MEMORY          1.32728
                CSSM1           CPUQ            14.63590
CSSM1           IOSYS1          DISKQ           7.32127
CSSM1           IOSYS2          DISKQ           7.31330
                CSSM1           FREEMEMORY      1.32728
                CSSM1           SETCMDTYPE      1.32861
                CSSM1           DECRCYCLES      14.63457

INVOCATION      INVOCATION      ELEMENT         MEAN QUEUE LENGTH
                                TERMINALSQ      21.30333
                CSSM1           MEMORY          3.69666
                CSSM1           CPUQ            0.45815
CSSM1           IOSYS1          DISKQ           0.56729
CSSM1           IOSYS2          DISKQ           0.53651

INVOCATION      INVOCATION      ELEMENT         STD. DEV. OF QUEUE LENGTH
                                TERMINALSQ      2.75469
                CSSM1           MEMORY          2.75471
                CSSM1           CPUQ            0.64897
CSSM1           IOSYS1          DISKQ           0.70332
CSSM1           IOSYS2          DISKQ           0.68847

INVOCATION      INVOCATION      ELEMENT         MEAN QUEUEING TIME
                                TERMINALSQ      15.66380
                CSSM1           MEMORY          2.78478
                CSSM1           CPUQ            0.03130
CSSM1           IOSYS1          DISKQ           0.07749
CSSM1           IOSYS2          DISKQ           0.07336
```

INVOCATION	INVOCATION	ELEMENT	STD. DEV. OF QUEUEING TIME
		TERMINALSQ	15.53620
	CSSM1	MEMORY	2.21000
	CSSM1	CPUQ	0.03351
CSSM1	IOSYS1	DISKQ	0.07348
CSSM1	IOSYS2	DISKQ	0.07022

INVOCATION	INVOCATION	ELEMENT	MEAN TOKENS IN USE
	CSSM1	MEMORY	35.57744

INVOCATION	INVOCATION	ELEMENT	MEAN TOTAL TOKENS IN POOL
	CSSM1	MEMORY	50.00000

INVOCATION	INVOCATION	ELEMENT	QUEUE LENGTH DISTRIBUTION
	CSSM1	MEMORY	0:0.08132
			1:0.15607
			2:0.17033
			3:0.15823
			4:0.10955
			5:0.08885
			6:0.06319
			7:0.04998
			8:0.05112
			9:0.03035
			10:0.02341
			11:0.01390
			12:3.2575E-03
			13:4.2857E-04

INVOCATION	INVOCATION	ELEMENT	QUEUEING TIME DISTRIBUTION
	CSSM1	MEMORY	1.00E+00:0.21800
			2.00E+00:0.50200
			3.00E+00:0.64700
			4.00E+00:0.75400
			5.00E+00:0.83500
			6.00E+00:0.91700
			7.00E+00:0.95000
			8.00E+00:0.96800

INVOCATION	INVOCATION	ELEMENT	MAXIMUM QUEUE LENGTH
		TERMINALSQ	25
	CSSM1	MEMORY	13
	CSSM1	CPUQ	2
CSSM1	IOSYS1	DISKQ	2
CSSM1	IOSYS2	DISKQ	2

INVOCATION	INVOCATION	ELEMENT	MAXIMUM QUEUEING TIME
		TERMINALSQ	132.74031
	CSSM1	MEMORY	13.42583
	CSSM1	CPUQ	0.38601
CSSM1	IOSYS1	DISKQ	0.66154
CSSM1	IOSYS2	DISKQ	0.70030

```
WHAT:
CONTINUE RUN:no
```

Having terminated both the performance measure dialogue and the simulation, we are now given the opportunity to define a new set of parameters and start a new run.

```
THINKTIME:
EXPANSION FINISHED.
```

A transcript of the dialogue with the EVAL command is available on mini-disk A with file name the same as the model name and file type RQ2PRNT. For example, we might now wish to print CSM RQ2PRNT A1 on a line printer.

It is also possible to embed model expansion and solution in a PL/I program. Users may define PL/I functions to provide numerical values to RESQ during the simulation run. For example, such a function might be used to read service times from a data file in order to implement a trace-driven simulation.

2. THE SETUP COMMAND

This section covers basic usage of the SETUP command within the CMS environment, the prompting mode of the SETUP command, the file mode of the SETUP command, the mixing of prompting and file modes of the SETUP command and the files used and produced by the SETUP command. Appendix 5 covers the error messages produced by the SETUP command.

2.1. SETUP Command with CMS

Before issuing the SETUP command, the user should be sure that his or her virtual machine has sufficient storage, that the virtual machine has access to the mini-disks containing the RESQ system files and the PL/I run time library, and that sufficient loader table space is provided. These steps typically will need to be taken only the first time RESQ is used, provided appropriate modifications are made to the CP directory and/or PROFILE EXEC.

To determine virtual storage currently available, issue the command

```
cp query virtual storage
```

which will produce a message of the form

```
STORAGE = 01024K
```

Usually 1024K (K = 1024 bytes) is sufficient for using the SETUP command. More than 1024K is often required for using the EVAL command (Section 13). To increase storage, enter the CP environment (e.g., by hitting the PA1 key on a 327X series terminal) and issue

```
cp define storage 1024k
```

The response to the DEFINE STORAGE command should be as with the QUERY STORAGE command, e.g.,

```
STORAGE = 01024K
```

However, if the CP directory maximum virtual storage entry does not allow the increase, the response will be

```
STORAGE EXCEEDS ALLOWED MAXIMUM
```

In this case it is necessary to have your CP directory maximum virtual storage entry changed (by the computer operations staff) in order to be able to successfully define the desired storage. (You may wish to have your CP directory default virtual storage entry changed to give you 1024K without issuing the DEFINE STORAGE command.) It is not strictly necessary to enter CP before issuing the DEFINE STORAGE command, but issuing the DEFINE STORAGE command from the CMS environment will produce an additional error message and leave the virtual machine in the CP environment. After defining sufficient storage, issue

```
ipl cms
```

followed by a blank line. This will restore the CMS environment and execute PROFILE EXEC.

To be sure the mini-disk containing the RESQ system files is available, issue

```
state setup exec *
```

If the RESQ files are available then this will only produce the normal CMS ready message. If the files are not available, the message

```
FILE 'SETUP EXEC' NOT FOUND
```

will be produced by the STATE command. To get access to the files, first determine the userid and virtual address of the mini-disk containing the RESQ files. Then issue the CP LINK and CMS ACCESS commands for this mini-disk. For example, if the RESQ files are on mini-disk 195 of userid Sauer, with password "abcde", then you might issue

```
cp link to Sauer 195 as 195 rr pass= abcde
access 195 b
```

(You may wish to insert lines such as these in your PROFILE EXEC.) The SETUP EXEC assumes that the PL/I optimizing compiler run time library is present on an accessed mini-disk and that the library has file name PLILIB and file type TXTLIB. The CMS STATE command may be used to verify that this is the case. If the library is not present, access to it must be obtained before using SETUP.

To determine whether sufficient pages are available for the CMS loader tables, issue

```
query ldrtbls
```

The response will be of the form

```
LDRTBLS  = 005
```

If the number of pages is less than 5, issue

```
set ldrtbls 5
```

to ensure sufficient pages are available. (You may wish to insert the SET LDRTBLS command in your PROFILE EXEC.)

The SETUP command may be issued without an argument, as in the example in Section 1. When issued without an argument, SETUP will prompt for a model name. Alternatively, SETUP may be issued with a single argument, which will be interpreted as the model name. Once the model name is established, the SETUP command is the same whether or not it was issued with an argument.

2.2. SETUP Command Prompting Mode

When the SETUP command is issued, it will look for a file with file name the same as the model name, file type RQ2INP and file mode A. If it finds such a file, it will treat this file as a dialogue file, using the dialogue file mode discussed in Section 2.3. If SETUP does not find such a dialogue file, it enters prompting mode, as in our example in Section 1.

SETUP examines only the first 72 characters of a line. Usually it is not necessary to have lines longer than this because of the repetition of prompts (e.g., the user can enter more than one class list per queue.) However, in some circumstances it may be necessary to create longer logical lines. If RESQ finds the string "++" at the end of a physical line, it assumes

that the next physical line is part of the current logical line, and the two lines are concatenated with the "++" removed. This concatenation of physical lines into a single logical line may be continued as long as the logical line does not exceed the internal buffer (see variable LINSIZ in file SETUPD RQ2DAT, Section 2.4). In producing dialogue files, it is sometimes necessary for SETUP to use the "++" concatenation because the length of the prompt plus the length of the reply exceeds 72 characters.

In prompting mode the special replies "how," "edit," "review," "save" and "quit" may be given in response to any SETUP prompt; their meanings are described below. These replies should not be included in dialogue files. (SETUP will not put these replies or any resulting dialogue in dialogue files.)

"How" is given by the user when a clarification of a prompt is desired. SETUP gives a short tutorial and then reissues the prompt.

"Edit" places the user in an editor looking at a dialogue file. This dialogue file is a transcript of the dialogue so far, excluding prompts receiving null replies and prompts receiving the five special replies. The user may make minor changes in this dialogue file, e.g., changing numeric values, or may make major changes, e.g., adding or deleting sections of dialogue. When the user leaves the editor (e.g., by filing) SETUP reprocesses the dialogue file left by the editor, as discussed in Section 2.3. (The user does not need to indicate to SETUP which file to process. SETUP will look for the RQ2INP file it gave to the user in the editor.) If the dialogue file is incomplete, as will usually be the case, then SETUP switches to prompting mode when it reaches the end of the file. (If the file is complete, SETUP exits without further prompting.)

The default editor used is the CMS EDIT command. However, if the user has a file "NORMAL EDITOR" on the A disk, then the first word in that file is assumed to be the name of an editor and that editor is used. For example, to use the VM/System Product Editor (XEDIT) with SETUP the NORMAL EDITOR file should have contents

```
XEDIT
```

This assumes that the System Product Editor is available on the specific CMS system being used. If the editor to be used is invoked by an EXEC, then "EXEC" should follow the EXEC name on the NORMAL EDITOR record. (Usually editors are invoked by CMS MODULE files.) For example, if you have an editor which you invoke using MYEDITOR EXEC, then the NORMAL EDITOR file should have contents

```
MYEDITOR EXEC
```

"Review" displays the dialogue file on the terminal so that the user may review what he or she has done. The dialogue file is a transcript of the dialogue so far, excluding prompts receiving null replies and prompts receiving the five special replies. The dialogue continues after the display with the prompt which received the "review" reply.

"Save" causes the dialogue to terminate, with the dialogue so far, both from file and interactive mode, saved in the dialogue file (with the same file name as the model name and file type RQ2INP).

"Quit" causes the dialogue file to terminate, with the last dialogue file retained on file type RQ2INP. This last file is as it existed after the last SETUP edit command, if there was one. Otherwise it is as it existed when SETUP was issued (possibly empty). What would

have become the new dialogue file if "save" had been issued is available with file type RQ2REC.

2.3. SETUP Command Dialogue File Mode

In dialogue file mode, the function of SETUP is analogous to that of a compiler for a programming language. After being given the model name, (in response to the MODEL: prompt, as an argument or implicitly after editing during prompting mode), SETUP will look for a file with the model name as file name, file type RQ2INP and file mode A. The file may have either fixed or variable length records up to 80 characters long. However, SETUP will only examine the first 72 characters of each record. Multiple physical records may be concatenated into a single logical line, as discussed in Section 2.2. If SETUP finds this file, then it will translate the file, issuing error messages as necessary, until it reaches the end of the file. If the file is syntactically complete, then SETUP will terminate without prompting the user. If the file is incomplete, then SETUP will switch to prompting mode to complete the dialogue.

A number of RESQ features are available in dialogue file mode which are not available in prompting mode. For example, parameters and identifiers with distribution values instead of numeric values may be defined, "global variables" for use during simulation may be defined, and certain simulation specific dialogues may be used. These particular features will be discussed in Sections 3 and 12, respectively. Perhaps, the most important RESQ feature available only in dialogue file mode is the "INCLUDE" statement.

The INCLUDE statement in RESQ is analogous to the preprocessor %INCLUDE statements of PL/I or PASCAL. The form of the INCLUDE statement is "INCLUDE:" followed by a file name. When SETUP encounters an INCLUDE statement, it searches for a file with the given file name; this file may either be a separate CMS mini-disk file or be a member of a MACLIB. If the file is located, then the entire text of the file is logically substituted in place of the INCLUDE statement and the text of the included file is processed by SETUP as if it had been part of the original model definition. The file specified in the INCLUDE statement must either have a file type of RQ2INP or be a member of a library with file type MACLIB SETUP will look for the file on all currently accessed mini-disks, not just mini-disk A. SETUP will first look for a separate CMS file with the specified file name and file type RQ2INP. If SETUP does not find such a file on any accessed mini-disk, then it will look in each MACLIB in the list (if any) of MACLIB's declared as global by issuing the CMS GLOBAL MACLIB command prior to issuing the SETUP command. (The MACLIB's are searched in the order listed in the GLOBAL command.) In either case, the file should have fixed length records with length 80. SETUP will only examine the first 72 characters of each record. If the file is not found, SETUP will issue an error message.

The INCLUDE statement is typically used to include submodel or queue type definitions. However, arbitrary portions of dialogue may be included with the INCLUDE statement; the INCLUDE facility is a general text substitution mechanism. An INCLUDE statement can occur almost anywhere in a dialogue file. Specifically, an INCLUDE statement can occur on any line in which a RESQ2 keyword prompt and reply can occur. An INCLUDE statement may not occur where SETUP would expect an identifier prompt for initializing an identifier or global variable or a keyword line where no reply occurs (e.g., "SERVER-").

A dialogue file can contain an arbitrary number of INCLUDE statements. It is possible to use INCLUDE statements in a nested manner; that is, a file to be included in a model can itself contain INCLUDE statements. Nesting of INCLUDE statements is allowed to a

maximum depth of 10. The dialogue parsed as a result of INCLUDE: statements does not appear in the RQ2INP file produced by SETUP.

In addition to the dialogue file (RQ2INP), SETUP produces a listing file with file name the same as the model name, file type RQ2LIST and file mode A. This file is very similar to the dialogue file, but it includes error messages at points where errors were detected (if any were detected), line numbers for each line, nesting levels of submodels and any dialogue parsed as a result of INCLUDE: statements. Following are some fragments of the RQ2LIST file for the example in Section 1.3.

```
RESQ2 Translator V2.04 (01/19/82)  Time: 16:55:48  Date: 01/29/82

*  1*  0*  MODEL:CSM
*  2*  0*      METHOD:simulation
*  3*  0*      NUMERIC PARAMETERS:thinktime users
*  4*  0*      NUMERIC IDENTIFIERS:userframes
*  5*  0*         USERFRAMES:50
*  6*  0*      MAX JV:1 /*0: command type, 1: cycle count*/
*  7*  0*      QUEUE:terminalsq
*  8*  0*         TYPE:is
*  9*  0*         CLASS LIST:terminals
* 10*  0*            SERVICE TIMES:thinktime
* 11*  0*      SUBMODEL:cssm /*Computer System Submodel*/
* 12*  1*         NUMERIC PARAMETERS:pageframes
* 13*  1*         CHAIN PARAMETERS:interactiv
          ...
* 35*  1*         SET NODES:decrcycles
* 36*  1*            ASSIGNMENT LIST:jv(cyclecount)=jv(cyclecount)-1
* 37*  1*         SUBMODEL:iosys
* 38*  2*            CHAIN PARAMETERS:interactiv
* 39*  2*            QUEUE TYPE:diskdef
* 40*  2*               NODE PARAMETERS:servicecls
          ...
* 55*  2*         END OF SUBMODEL IOSYS
* 56*  1*         INVOCATION:iosys1
* 57*  1*            TYPE:iosys
          ...
* 68*  1*      END OF SUBMODEL CSSM
* 69*  0*      INVOCATION:cssm1
* 70*  0*         TYPE:cssm
          ...
* 92*  0*      LIMIT - CP SECONDS:5
* 93*  0*      SEED:1
* 94*  0*      TRACE:no
* 95*  0*  END

 NO FATAL ERRORS DETECTED DURING COMPILATION.
```

2.4. SETUP Command Files

We have already discussed or mentioned most of the files used or produced by the SETUP command. The normal input to the SETUP command is from three files: (1) SYSIN - the SETUP EXEC issues a CMS FILEDEF command defining SYSIN to be the terminal.

(2) The dialogue file (RQ2INP) if one exists and (3) SETUPD RQ2DAT, which is used to define the sizes of certain internal tables. SETUP cannot determine in advance the appropriate sizes for its symbol, expression and routing tables. It cannot determine in advance the appropriate size for its buffers for storing a logical line. File SETUPD RQ2DAT on the mini-disk containing SETUP EXEC contains sizes for these tables and buffers. The default content of the file is

```
SYMSIZ=1005 , EXPSIZ=2005 , ELVSIZ=2505 , RTBSIZ=1005 , LINSIZ=1729;
/*DIMENSIONS OF SYMTAB, EXP. TAB, ELEMENT VECTOR, ROUTING TAB, BUFFERS*/
```

where SYMSIZ is the maximum number of symbols (identifiers), EXPSIZ is the maximum number of expressions, ELVSIZ is the maximum number of expression components (e.g., 3.1*(i−3) has 5 components: 3.1, *, i, − and 3), RTBSIZ is the maximum number of routing transitions and LINSIZ is one more than the maximum length (in characters) of a logical line. The user may have a copy of SETUPD RQ2DAT on a mini-disk in the search order before the mini-disk containing the SETUP EXEC, to be used instead of the default copy. The user may increase (or decrease within reason) these table and buffer sizes in this copy of SETUPD RQ2DAT.

While executing, the SETUP command produces four files: (1) SYSPRINT - the SETUP EXEC issues a CMS FILEDEF command defining the terminal to be SYSPRINT. (2) RQ2REC - this is the file which normally will become RQ2INP at the end of the SETUP command (unless the "quit" special reply is used). (3) RQ2LIST and (4) RQ2COMP - this is the file, with file name the same as the model name and file type RQ2COMP, which will provide the input to the EVAL command. If SETUP is used in dialogue file mode and discovers errors, it will erase the RQ2COMP file it has generated. Unless the "quit" special reply is used, SETUP will erase the RQ2INP file it was given and rename the RQ2REC file it generated to be the new RQ2INP.

Figure 2.1 shows these files and their relationships with the commands.

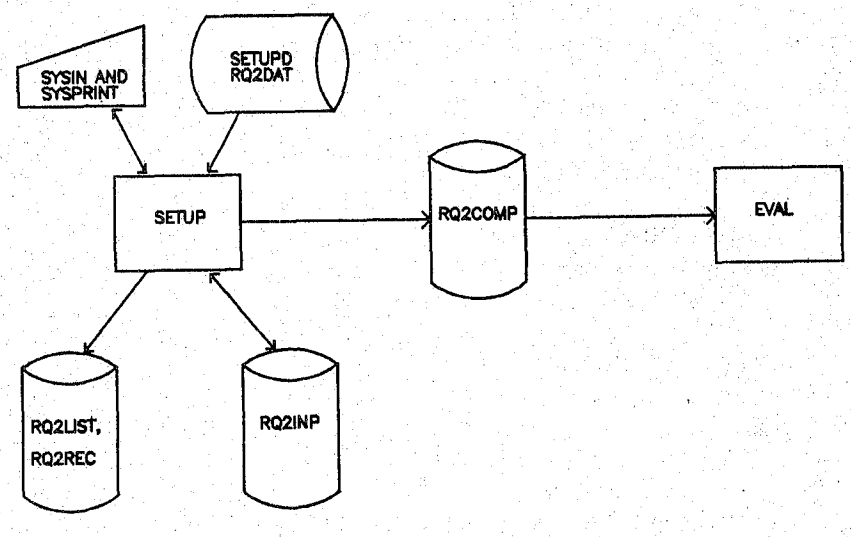

Figure 2.1 - Files used with SETUP

3. PARAMETERS, IDENTIFIERS, VARIABLES AND ARRAYS

This section covers the syntax and semantics of the declarations of parameters, identifiers, variables, chain arrays and node arrays at the beginning of either a model or submodel. The syntax and semantics are the same in either case, except where otherwise noted. (Some similar declarations are used with queue type definitions, but the differences are significant enough that we discuss those declarations separately in Section 6.) This section also covers the syntax and semantics of the declaration of the extents of the vectors of job and chain variables. Some of the declarations are not possible in interactive mode because no prompts are issued to give the opportunity to make the declarations. All of these declarations are optional in dialogue files (assuming the declared elements are not needed). We discuss these declarations in the order they may appear in a dialogue file.

3.1. Parameters

There are four types of parameters allowed in RESQ, numeric parameters, distribution parameters, node parameters and chain parameters. Node parameters and chain parameters are allowed only in submodels.

Numeric parameters defined at the beginning of a model are given constant numerical values (internally represented in floating point) when the EVAL or EVALT commands (Section 13) are issued or when the appropriate procedure is called from a PL/I program calling RESQ (Section 14). Numeric parameters defined at the beginning of a submodel are given numerical values when the submodel is invoked (Section 10). Names of numeric parameters may be used in place of numerical constants anywhere in the SETUP dialogue that numerical constants are appropriate. Numeric parameters may be scalars, vectors or matrices. As illustrated in Section 1.3, the syntax consists of "NUMERIC PARAMETERS:" followed by a list of one or more names to be used for the parameters. A vector parameter is declared by following the name by a parenthesized expression for the number of elements in the vector. The elements are indexed starting at one (1). The expression may be any expression which (1) can be evaluated at this point in the dialogue, e.g., any parameters in the expression have been previously declared, and (2) is independent of simulation (see Appendix 3 for clarification of this distinction). A matrix parameter is declared by following the name by a left parenthesis ("("), an expression for the number of rows, a semi-colon (";"), an expression for the number of columns and a right parenthesis (")"). The rows and columns are indexed starting at one (1). The same constraints are placed on the expressions as for the expression giving the number of elements in a vector. The line declaring numeric parameters may be repeated as many times as necessary to declare the desired parameters. However, in declaring parameters, the user should keep in mind the effort required to define values for parameters, e.g., when the EVAL command is issued, and consider using identifiers instead. Following is an example of numeric parameter declaration:

```
NUMERIC PARAMETERS:a b(a)
NUMERIC PARAMETERS:c(2;a+1) d
```

Distribution parameters may be defined only within dialogue files. Distribution parameters defined at the beginning of a model are given values representing probability distributions when the EVAL or EVALT commands are issued or when the appropriate procedure is called from a PL/I program calling RESQ (Section 14). Distribution parameters defined at the beginning of a submodel are given values representing probability distributions when the submodel is invoked (Section 10). Names of distribution parameters may be used in place of probability distributions anywhere in the SETUP dialogue that probability distributions are appropriate. The values given for distribution parameters may be either numerical values or

probability distributions. RESQ2 probability distributions are discussed in Appendix 3. A numerical value given for a distribution parameter will be interpreted as either (1) the mean of an exponential distribution, where a continuous distribution is expected, e.g., for service times, or (2) a constant distribution, where a discrete distribution is expected, e.g., for numbers of tokens to be allocated or in set node assignment statements. Distribution parameters may be scalars, vectors or matrices. The syntax consists of "DISTRIBUTION PARAMETERS:" followed by a list of one or more names to be used for the parameters. Vector and matrix parameters are declared as with numeric parameters. The line declaring distribution parameters may be repeated as many times as necessary to declare the desired parameters. Following is an example of distribution parameter declaration:

```
DISTRIBUTION PARAMETERS:b
DISTRIBUTION PARAMETERS:c(2;a+1) d
```

Node parameters may be defined only within submodels. Node parameters are used to allow a submodel to refer to nodes outside of the submodel. For further discussion of the use of node parameters, see Section 10 and Appendix 1. Names of node parameters may only be used in routing definitions and as arguments to status functions. Node parameters may be scalars or vectors. The syntax consists of "NODE PARAMETERS:" followed by a list of one or more names to be used for the parameters. Vector parameters are declared as with numeric parameters. The line declaring node parameters may be repeated as many times as necessary to declare the desired parameters. Following is an example of node parameter declaration:

```
NODE PARAMETERS:b
NODE PARAMETERS:c(2*a) d
```

Chain parameters may be defined only within submodels. Chain parameters are used to allow routing chains to cross submodel boundaries, i.e., to connect nodes inside and outside of submodels. For further discussion of the use of chain parameters, see Section 10 and Appendix 1. Names of chain parameters are only used in routing definitions in response to the "CHAIN:" prompt. Chain parameters are always given type "external" within a submodel. Chain parameters may be scalars or vectors. The syntax consists of "CHAIN PARAMETERS:" followed by a list of one or more names to be used for the parameters. Vector parameters are declared as with numeric parameters. The line declaring chain parameters may be repeated as many times as necessary to declare the desired parameters. Following is an example of chain parameter declaration:

```
CHAIN PARAMETERS:b
CHAIN PARAMETERS:c(2*a) d
```

3.2. Identifiers

There are two types of identifiers allowed in RESQ, numeric identifiers and distribution identifiers.

Numeric identifiers are given numeric expressions defining their values (internally represented in floating point) immediately following their declarations. Names of numeric identifiers may be used in place of numerical constants anywhere in the SETUP dialogue that numerical expressions are appropriate. However, simulation dependent values (see Appendix 3) may not be used in these expressions. Numeric identifiers may be scalars, vectors or matrices. As illustrated in Section 1.3, the syntax consists of "NUMERIC IDENTIFIERS:" followed by a list of one or more names to be used for the identifiers. Names of vectors and matrices are declared as with numeric parameters. Immediately following the line declaring

the names of the identifiers will be one or more lines (one per name) giving the identifier name, a colon (":") and the defining expression(s) for that name. In the case of a vector, defining expressions for all elements are given on the same line. If there are fewer expressions than the number of elements in the vector, the last expression is also used for the remaining elements. Matrices are stored internally as vectors, by rows, i.e., if a matrix has m rows and n columns, the matrix is stored as a vector with $m \times n$ elements, with the first n elements of the internal vector containing the first row of the matrix, the second n elements of the internal vector containing the second row of the matrix, and so on. Defining expressions for all elements of a matrix are given on the same logical line (using concatenation of physical lines, if necessary) to specify the elements of the internal vector representation. If there are fewer expressions than the number of elements in the matrix, the last expression is also used for the remaining elements. The line declaring numeric identifiers may be repeated as many times as necessary to declare the desired identifiers. Following is an example of numeric identifier declaration:

```
NUMERIC IDENTIFIERS:a b(3) c(3;2)
   A:3.1*min(d,1)
   B:0
   C:14.1 7 13
```

In this example, all three elements of b are zero, c(1;1) is 14.1; c(1;2) is 7 and the remaining elements are 13.

Distribution identifiers may be defined only within dialogue files. Distribution identifiers correspond to numeric identifiers as distribution parameters correspond to numeric parameters; the syntax is the same except for the keyword difference ("DISTRIBUTION" instead of "NUMERIC"). The defining expressions given for distribution identifiers may include, but need not include, references to probability distributions. Names of distribution identifiers may be used in place of probability distributions anywhere in the SETUP dialogue that probability distributions are appropriate. (RESQ2 probability distributions are discussed in Appendix 3.) Distribution identifiers may be scalars, vectors or matrices. Where a distribution identifier is used in the dialogue, the effect is as if the defining expression for that distribution identifier (or identifier element, in case of a vector or matrix identifier) were used in that place, except for possible differences due to the scope of names with respect to submodels, i.e., the names in effect where the defining expression is given are the names referenced in the expression evaluation. Following is an example of distribution identifier declaration:

```
DISTRIBUTION IDENTIFIERS:a b(2;3) c
   A:discrete(1,.8;2,.15;3,.05)
   B:discrete(x,.8;y,.2) z+bE(1,0;1,1) 3
   C:propagate+standard(leng,0)/capacity
```

3.3. Global Variables

Global variables may be declared only in dialogue files. Global variables in RESQ correspond to variables in programming languages and can be used for essentially the same purposes. The term "global" is used to distinguish these variables from job variables (JV) and chain variables (CV). However, global variables may be local to submodels in the sense of the scope of the names of the variables. Global variables are internally represented in double precision floating point. The declarations and initial values of global variables are given using the same syntax as used to declare and define values for numeric identifiers, except the keywords "GLOBAL VARIABLES" are used instead of "NUMERIC IDENTIFIERS". Simulation dependent values (see Appendix 3) may be used in the defining expressions for

initial values for global variables. Global variables may be scalars, vectors or matrices. The same conventions for internal storage of vector and matrix elements that are used with numeric identifiers are used with global variables. After global variables are declared, the values may be changed by assignments associated with set nodes. Examples of the use of global variables are given in Appendix 1. Following is an example of global variable declaration:

```
GLOBAL VARIABLES:a b(3) c(3;2)
   A:3.1*min(d,1)
   B:a discrete(1,.3;2,.7) 0
   C:14.1 7 13
```

The expressions are evaluated before simulation begins, so simulation dependent expressions involving status functions (Appendix 3) are not particularly useful in these expressions. Distributions may be useful and the USER function (Appendix 3) may be useful. The second element of b in the example will either be 1 or 2, depending on the random number generated in evaluating the discrete distribution.

3.4. Chain and Node Arrays

Arrays of routing chains are useful where the several chains have substantially the same characteristics and the differences between the chains can be simply characterized. Chain arrays will always be vectors. The routing for an array of chains may be specified for the entire array in a single chain definition section. The use of chain arrays implies the use of node arrays with the same numbers of elements in the chain and node arrays. The differences between chains are specified by use of numeric vectors or matrices in definitions of routing probabilities and predicates, in definitions of the node characteristics, etc. Names are declared as names of chain arrays in the declarations section of a model or submodel. The definition of the routing chains having this array name is given in the routing section as with scalar chains (Section 9). The declaration syntax consists of "CHAIN ARRAYS:" followed by a list of one or more names. Each name is followed immediately by a parenthesized expression for the number of elements in the array. The constraints on this expression are the same as for expressions for the number of elements in a numeric parameter vector. The line declaring chain arrays may be repeated as many times as necessary to declare the desired arrays. Following is an example chain array declaration:

```
CHAIN ARRAYS:interactiv(no_groups) batch(no_types)
```

Arrays of nodes are necessary with chain arrays and are useful in other situations. Node arrays will always be vectors. All elements of a node array will have the same node type, e.g., if one element of a particular array is a class, then all elements of that array will be classes. Further, if one element of a particular node array belongs to a particular queue, then all elements of the array belong to that queue. Names are declared as names of node arrays in the declarations section of a model or submodel. The definition of the nodes having this array name is given in the same way as with scalar nodes (Sections 4-8). The declaration syntax consists of "NODE ARRAYS:" followed by a list of one or more names. Each name is followed immediately by a parenthesized expression for the number of elements in the array. The constraints on this expression are the same as for expressions for the number of elements in a numeric parameter vector. The line declaring node arrays may be repeated as many times as necessary to declare the desired arrays. Following is an example node array declaration:

```
NODE ARRAYS:terminals(no_groups) batch_s(no_types)
NODE ARRAYS:cpu(no_groups+no_types)
```

3.5. Extents of Job and Chain Variables

Each job in RESQ has a vector of job variables (JV) stored with that job. The vector begins with index 0 and has extent as declared by the MAX JV statement, as illustrated in Section 1.3. (The disparity of starting with index 0 for job variables and chain variables and with index 1 for all other RESQ vectors is due to preservation of compatibility with early versions of RESQ.) The syntax is "MAX JV:" followed by an expression for the extent, where the expression has the same constraints as those for the number of elements in a numeric parameter vector. If the MAX JV statement is omitted, the value 1 is used for the extent, i.e., reference may only be made to JV(0) and JV(1). Except for jobs produced by split and fission nodes (Section 8), all job variables are initialized to have value 0 (zero). See Section 13 for a discussion of the storage requirements of jobs and job variables.

Each chain in RESQ has a vector of chain variables (CV) stored with that chain. The vector begins with index 0 and has extent as declared by the MAX CV statement. Chain variables may be used to affect the arrival times for jobs in open chains (Section 9). Otherwise, it is usually advisable to use global variables instead of chain variables. The syntax is "MAX CV:" followed by an expression for the extent, where the expression has the same constraints as those for the number of elements in a numeric parameter vector. If the MAX CV statement is omitted, the value 0 is used for the extent, i.e., reference may only be made to CV(0). All chain variables are initialized to have value 1 (one).

4. ACTIVE QUEUES

This section covers the syntax and semantics of the definitions of active queues using predefined queue types. Section 6 covers definition of queues with user defined queue types. A job's activity is typically focused on the resources of active queues. A job typically has no interaction with other model elements while at an active queue. An active queue consists of one or more servers and one or more waiting lines called "classes". A class belonging to one active queue may not belong to another active queue. The classes categorize the characteristics of jobs currently at the queue in terms of work demand (service requirement) distributions, priorities and routing. (A class is a particular kind of node in the sense of RESQ routing from node to node.) Examples of work demand could be number of instructions, number of bytes, etc. In general, work demanded is divided by service rate to obtain service time. The service rate is the amount of work the server can perform in one unit of time. In the common special case of fixed rate servers, the server may be assumed to have unit rate of service and the work demand may be expressed as service time. Jobs within a class may be further distinguished, e.g., by the values of job variables. A job arriving at a class makes a request for service and waits until it is assigned a server. Once the job is assigned a server, it receives service from that server until the service request is satisfied. The service may be preempted by other jobs arriving at the queue or the server may be shared with other jobs, depending on the queueing discipline.

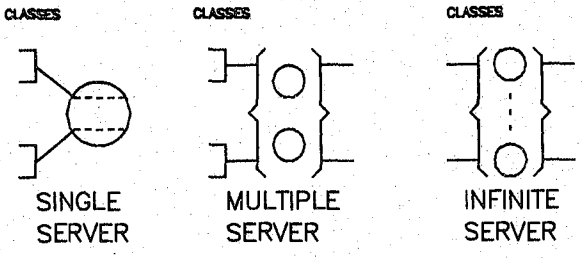

Figure 4.1 - Active Queues
Single, Multiple, Infinite Server

Figure 4.1 (a duplicate of Figure 1.2) shows the diagram symbols used for active queues. We first discuss the simple predefined queue types and then discuss the general case. The discussion presumes that simulation is used for model solution. See Section 11 for restrictions for numerically solved models.

4.1. The FCFS Queue Type

The FCFS queue type is used to define a single server queue with first come first served scheduling. The server has a fixed rate of service of one unit of work per unit of simulated time. Jobs are served in order of arrival at the queue, i.e., class distinctions are ignored for scheduling purposes. After the queue type specification, the FCFS type definition consists of one or more pairs of lines, the first element of the pair being a list of classes followed by a list of service time distributions. The following example illustrates the FCFS type:

```
QUEUE:q
   TYPE:fcfs
   CLASS LIST:          a    b                  c
      SERVICE TIMES:.5       user(jv(0);3)      discrete(10,.5;20,.5)
   CLASS LIST:          d    e(*)  /*"(*)" is optional*/
      SERVICE TIMES:.6+standard(jv(3),0)
```

```
   CLASS LIST:        g(*) /*"f" is a reserved keyword for "false"*/
      SERVICE TIMES: h(*)
```

The line pairs of class lists and service times may be repeated as many times as necessary to define the required classes. An element in the list of service time distributions may be any numerical expression, possibly including simulation dependent values such as status functions and job variables and possibly including distribution parameters and identifiers. A single expression may be given instead of one expression per element in the class list; the single expression is then used for all classes in the list.

An element in the class list may be the name of a node array, indicating all elements of the array. The node array name may be followed by "(*)" to explicitly indicate all elements are listed, but individual elements of a node array may not be listed. An element in the service time distribution list corresponding to a node array in the class list must be a vector (e.g., a numeric parameter or a distribution identifier) of the same length, unless the service time distribution list consists of a single expression to be used for all elements in the class list.

If a service time expression, after resolution of parameters and identifiers, contains no references to RESQ probability distribution keywords, then the value of the expression is interpreted as the mean of a (negative) exponential probability distribution. (The RESQ probability distribution keywords are BE, DISCRETE, STANDARD and UNIFORM. See Appendix 3 for further discussion of distributions.) If the expression does contain at least one distribution keyword, then the expression is used directly. In either case, when a job arrives at a class, a sample is obtained from the service time distribution and stored with the job to be used when the job is assigned a server.

4.2. The IS Queue Type

The IS queue type is used to define an infinite server "queue." (Since there is no waiting for service, scheduling is not an issue.) Each server serves at a fixed rate of one unit of work per unit of simulated time. After the queue type specification, the IS type definition consists of one or more pairs of lines, the first element of the pair being a list of classes followed by a list of service time distributions. The rules for the class lists and service time lists are the same as with the FCFS queue type. The following example illustrates the IS type:

```
QUEUE:q
   TYPE:is
   CLASS LIST:        a    b                c
      SERVICE TIMES:.5    user(jv(0);3)    discrete(10,.5;20,.5)
   CLASS LIST:        d    e(*)  /*"(*)" is optional*/
      SERVICE TIMES:.6+standard(jv(3),0)
   CLASS LIST:        g(*)
      SERVICE TIMES: h(*)
```

4.3. The PS Queue Type

The PS queue type is used to define a single server queue with the processor sharing queueing discipline. Processor sharing is the limiting case of a round robin ("time slicing") discipline when the quantum ("time slice") tends to zero, provided there is negligible overhead in switching from job to job. The server serves at a fixed rate of one unit of work per unit of simulated time. After the queue type specification, the PS type definition consists of one or more pairs of lines, the first element of the pair being a list of classes followed by a list of

service time distributions. The rules for the class lists and service time lists are the same as with the FCFS queue type. The following example illustrates the PS type:

```
QUEUE:q
   TYPE:ps
   CLASS LIST:        a    b                c
      SERVICE TIMES:.5    user(jv(0);3)    discrete(10,.5;20,.5)
   CLASS LIST:        d    e(*)  /*"(*)" is optional*/
      SERVICE TIMES:.6+standard(jv(3),0)
   CLASS LIST:        g(*)
      SERVICE TIMES: h(*)
```

4.4. The LCFS Queue Type

The LCFS queue type is used to define a single server queue with the last come first served preemptive resume queueing discipline. An arriving job always preempts a job in service, if there is one. Jobs are served in reverse order of arrival. When a job is preempted and later restarted, its remaining service request is the original request less any service already received. The server serves at a fixed rate of one unit of work per unit of simulated time. After the queue type specification, the LCFS type definition consists of one or more pairs of lines, the first element of the pair being a list of classes followed by a list of service time distributions. The rules for the class lists and service time lists are the same as with the FCFS queue type. The following example illustrates the LCFS type:

```
QUEUE:q
   TYPE:lcfs
   CLASS LIST:        a    b                c
      SERVICE TIMES:.5    user(jv(0);3)    discrete(10,.5;20,.5)
   CLASS LIST:        d    e(*)  /*"(*)" is optional*/
      SERVICE TIMES:.6+standard(jv(3),0)
   CLASS LIST:        g(*)
      SERVICE TIMES: h(*)
```

4.5. The PRTY Queue Type

The PRTY queue type is used to define a single server queue with a nonpreemptive priority queueing discipline. An arriving job is assigned a positive integer priority. (This priority is then fixed until the job leaves the queue.) A priority value closer to zero is considered a higher priority than a priority value farther from zero. When the server becomes available and there are waiting jobs, a job with the smallest priority value is selected for service. Scheduling is first come first served among jobs with the same priority value. The server serves at a fixed rate of one unit of work per unit of simulated time. After the queue type specification, the PRTY type definition consists of one or more triples of lines, the first element of a triple being a list of classes, the second a list of service time distributions and the third a list of priority expressions. The rules for the class lists and service time lists are the same as with the FCFS queue type. The syntax of the priority expression lists is the same as with service time lists. If a priority expression does not evaluate to an integer, the value is truncated to an integer value (the fraction is discarded). The following example illustrates the PRTY type:

```
QUEUE:q
   TYPE:prty
```

```
CLASS LIST:        a    b                c
   SERVICE TIMES:.5    user(jv(0);3)    discrete(10,.5;20,.5)
   PRIORITIES:     5    1                jv(j_prty)
CLASS LIST:        d    e(*)  /*"(*)" is optional*/
   SERVICE TIMES:.6+standard(jv(3),0)
   PRIORITIES:     3
CLASS LIST:        g(*)
   SERVICE TIMES: h(*)
   PRIORITIES:     p(*)
```

4.6. The PRTYPR Queue Type

The PRTYPR queue type is used to define a single server queue with a preemptive priority queueing discipline. Preemption decisions are based on the differences between the priority of the job being served and the priorities of the other jobs in the queue and on the preemption distance specified in the queue definition. By appropriate choice of priority expressions and preemption distance, very general scheduling mechanisms can be represented with the PRTYPR queueing discipline. An arriving job is assigned a positive integer priority. (This priority is then fixed until the job leaves the queue.) A priority value closer to zero is considered a higher priority than a priority value farther from zero. If the arriving job is of higher priority than the job in service (if there is a job in service), then if the difference between the priority values is at least the preemption distance, then preemption occurs. When a job is preempted and later restarted, its remaining service request is the original request less any service already received. A preemption distance of 1 (one) results in a strictly preemptive discipline, i.e., preemption always occurs when a higher priority job arrives, and a sufficiently large preemption distance, e.g., 2147483647 ($2^{31}-1$), results in a strictly nonpreemptive discipline as with PRTY. When the server becomes available and there are waiting jobs, a job with the smallest priority value is selected for service. Scheduling is first come first served among jobs with the same priority value. The server serves at a fixed rate of one unit of work per unit of simulated time. After the queue type specification, there is a line with "PREEMPT DIST:" followed by an expression for the preemption distance. This expression must be simulation independent (see Appendix 3). After that line the PRTYPR type definition consists of one or more triples of lines, the first element of a triple being a list of classes, the second a list of service time distributions and the third a list of priority expressions. The rules for the class lists and service time lists are the same as with the FCFS queue type. The syntax of the priority expression lists is the same as with service time lists. If a priority expression does not evaluate to an integer, the value is truncated to an integer value (the fraction is discarded). The following example illustrates the PRTYPR type:

```
QUEUE:q
   TYPE:prtypr
   PREEMPT DIST:3
   CLASS LIST:        a    b                c
      SERVICE TIMES:.5    user(jv(0);3)    discrete(10,.5;20,.5)
      PRIORITIES:     5    1                jv(j_prty)
   CLASS LIST:        d    e(*)  /*"(*)" is optional*/
      SERVICE TIMES:.6+standard(jv(3),0)
      PRIORITIES:     3
   CLASS LIST:        g(*)
      SERVICE TIMES: h(*)
      PRIORITIES:     p(*)
```

For discussion purposes assume that jobs arriving at class C have JV(J__PRTY) with value 13 and that all elements of P(*) are 10. Jobs arriving at class B would preempt jobs at classes A, C and G(*) but not jobs at classes D and E(*). Jobs arriving at classes D or E(*) would preempt jobs at classes C and G(*) but not class A. Jobs arriving at class A would preempt jobs at class C or classes G(*). Jobs arriving at classes G(*) would preempt jobs at class C.

4.7. The ACTIVE Queue Type

The ACTIVE queue type is used to define active queues not definable with the above queue types. This general case allows declaration of multiple server queues other than IS queues, queueing disciplines not allowed by the other predefined queue types, and queues with servers that have queue length dependent service rates and/or with servers that will accept jobs from only a subset of the classes of the queue. Several of the lines in the ACTIVE dialogue are optional in dialogue files or apply only to certain queueing disciplines. After the queue type specification, there is a line for definition of the number of servers, which consists of "SERVERS:" followed by an expression for the number of servers. The expression must be simulation independent. If the number of servers line is omitted, the queue will have a single server by default.

Next is a required line for definition of queueing discipline which consists of "DSPL:" followed by a keyword representing a queueing discipline. The allowed keywords are FCFS, PS, LCFS, PRTY, PRTYPR, SRTF and LRTF. (In addition to the generality of scheduling mechanisms possible with the PRTYPR discipline as mentioned in Section 4.6, it should be recognized that set nodes, routing predicates, passive queues and other RESQ elements may be used to build complex scheduling mechanisms, as illustrated in Appendix 1.) Only FCFS, PS and PRTY are allowed with multiple server queues. With SRTF (shortest remaining time first), the job chosen for service is always the one with the shortest remaining service time. If an arriving job has service time less than the remaining service time of the job in service, if any, then the job in service is preempted (with its remaining time saved so that it can resume where it left off) and the arriving job gets the server. SRTF is the optimum discipline for a queue in isolation in the sense that it minimizes the mean queueing time by maximizing the number of completed queueing times. With LRTF (longest remaining time first), the job chosen for service is always the one with the longest remaining service time. A job is not preempted by jobs already in the queue even though its remaining service time has become shorter than one of those jobs because of the progress it has made. (Thus LRTF is not the worst possible discipline in the sense that SRTF is the optimal discipline.) If an arriving job has service time more than the remaining service time of the job in service, if any, then the job in service is preempted (with its remaining time saved so that it can resume where it left off) and the arriving job gets the server. With the regenerative method for confidence intervals (Section 12), the regeneration state should not have any jobs at queues with the SRTF and LRTF disciplines. If the queueing discipline is PRTYPR, then after the queueing discipline line there is a required line giving the preemption distance, as with the line following PRTYPR with the PRTYPR predefined queue type.

Next comes one or more pairs or triples of lines containing class lists and class characteristics, depending on the queueing discipline. There will be pairs of lines for FCFS, PS, LCFS, SRTF and LRTF and triples of lines for PRTY and PRTYPR. The first element of a pair will be a class list, as with the FCFS predefined type. The second element of a pair will be a list of work demand expressions analogous to the service time expressions of the FCFS predefined type. "WORK DEMANDS:" is used instead of "SERVICE TIMES:" because the service time will be determined by both the work demand and the service rate, i.e., the amount of work requested by the job will be divided by the service rate of the server to determine service time. Except for the change in keywords, the characteristics are the same for the work demand line

in the ACTIVE type and the service time line in the FCFS type. In the case of PRTY and PRTYPR queueing disciplines, the class list and work demand lines are followed by a list of priority expressions, as with the PRTY and PRTYPR predefined types.

After the class definition section comes the optional server definition section. If no server definitions are given, then all servers are assumed to be fixed unit rate servers which accept jobs of all classes of the queue. Server definitions are primarily useful where servers are to have service rate dependent on queue length, i.e., the total number of jobs at the queue and/or where servers only accept jobs from a subset of the classes of the queue.

Service rates dependent on queue length can be used to get the effect of multiple servers (though it is usually more efficient to actually define multiple servers), to represent increased or decreased server efficiency with varying queue lengths and/or to represent a subnetwork by a single "composite" queue in an approximate solution. Where service rates depend on queue length, rates are redetermined whenever the queue length changes. This is true for both simulation and numerical solution. If in a simulation it is desired that service times depend on queue length on arrival of a job, but are not affected by later changes in queue length (including arrivals during service) this can be accomplished by use of the QL or TQ status functions (see Appendix 3) in expressions for service times (or work demands). Queues with several classes may be defined with several servers such that each server will accept jobs from only a subset of the classes. Such a queue may be useful in representing multi-processor systems where the processors have different characteristics, e.g., some processors cannot initiate I/O.

A server definition consists of a line "SERVER -" followed by optional definitions of service rates and classes accepted. Service rates are defined by one or more lines of the form "RATES:" followed by a list of expressions giving service rates for different queue lengths. The expressions must be simulation independent (Appendix 3). The first rate given is used for queue length 1, the second rate (if given) is used for queue length 2, the third rate for queue length 3 and so on. The last rate given is used for all larger queue lengths as well as for that particular queue length. If no rates are given then the server will have a unit rate for all queue lengths. Classes accepted are declared by one or more lines of the form "ACCEPTS:" followed by a list of classes accepted by the server. The keyword ALL or an empty list may be used instead of a list of all classes of the queue. Similarly, if the classes accepted declaration is omitted, then all classes are accepted by the server. The number of server definitions may be smaller or larger than the number of servers specified at the beginning of the queue definition. If the number of definitions is smaller than the last definition, then the last definition is repeated to make up for the missing definition. Excess definitions are ignored. The numerical components of RESQ require that a queue with queue length dependent service rates has the same rate list for each of its servers. Class restricted servers are not allowed with the numerical solution components of RESQ.

Following is an example ACTIVE queue definition:

```
QUEUE:q
   TYPE:active
   SERVERS:2
   DSPL:fcfs
   CLASS LIST:c1 c2
      WORK DEMANDS:stime
   SERVER -
      RATES:1 /*1 job at the queue*/
      RATES:.9 /*2 or more jobs at the queue*/
   SERVER -
```

```
RATES:1 2
ACCEPTS:c2
```

5. PASSIVE QUEUES

This section covers the syntax and semantics of the definitions of passive queues using predefined queue types. Section 6 covers definition of queues with user defined queue types. Passive queues are not allowed with numerical solution (Section 11). Passive queues allow convenient representation of simultaneous resource possession. A job typically acquires tokens of a passive queue and holds on to them while visiting other queues (active and/or passive queues) and model elements. The job explicitly releases or destroys its tokens when it no longer needs them. A second use of passive queues is to model mechanisms such as communication protocols and protocols for channel-device interaction. (Such usage may involve other RESQ elements. See Appendix 1 examples.) A third use of passive queues is in measuring response times in subnetworks. The "queueing time" (response time) for a passive queue is defined as the time between a job's request for tokens of the passive queue and that job's freeing or destroying of the tokens.

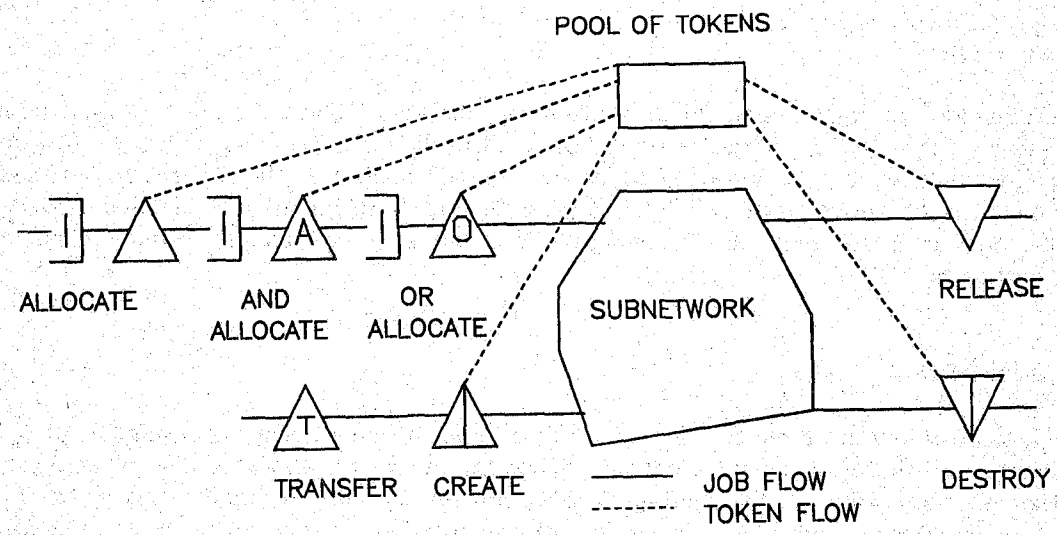

Figure 5.1 - Passive Queue

A passive queue consists of a pool of tokens to be allocated to jobs and a set of nodes which operate on that pool and the jobs holding tokens. In addition to the allocate and release nodes used in the example of Section 1, passive queues may have AND allocate nodes for simultaneous allocation from several different queues, OR allocate nodes for allocation from any one of several passive queues, transfer nodes for passing tokens back and forth between related jobs (Section 8), destroy nodes for destroying tokens held instead of releasing tokens and create nodes for adding new tokens to the pool. Except for AND and OR allocate nodes, a node belonging to one passive queue may not belong to another passive queue. AND and OR allocate nodes are usually associated with several passive queues. Fusion nodes (Section 8) and sinks (Section 9) may have the effect of releasing tokens. Figure 5.1 (a duplicate of Figure 1.3) shows the diagram symbols used to represent passive queues.

The passive queue definition dialogue begins with the line for the queue name, as in an active queue definition. Next is the line "TYPE:passive". There are no predefined types for passive queues other than the general case. Next is a required line giving the number of tokens initially in the pool. This line consists of "TOKENS:" followed by an expression. This

expression must be simulation independent and non-negative. Next is a required line giving the queueing discipline. This consists of "DSPL:" followed by a keyword representing the discipline, either FCFS, FF or PRTY.

For the moment our discussion of queueing disciplines with passive queues assumes no AND allocate nodes. The effects of those nodes will be discussed in Section 5.2. FCFS and PRTY are essentially the same with passive queues as with active queues. With FCFS, available tokens are allocated to jobs in order of arrival of the jobs at the queue, the earliest arrivals getting tokens first. With PRTY, priority expressions are associated with allocate nodes. Token assignment is in priority order, with first come first served used among jobs of the same priority. FF (first fit) is like FCFS but with one difference. If a job near the front of the queue in first come first served order requires more tokens than are available while a job further back (a later arrival) requires no more tokens than are available, with FCFS the second job will wait while with FF the second job will be given the tokens it requires.

The following example illustrates the beginning of a passive queue definition:

```
QUEUE:q
   TYPE:passive
   TOKENS:2
   DSPL:prty
```

The remainder of the passive queue definition consists of one or more sections defining particular types of nodes. Each section is optional in dialogue file mode, but a passive queue must have at least one allocate node (possibly an AND or an OR allocate node). In interactive mode, only the sections for "plain" allocate nodes, release nodes, destroy nodes and create nodes are presented. The following sections discuss these dialogue sections in the order expected by SETUP.

5.1. Allocate Nodes

A job arriving at an allocate node (a "plain" allocate node) requests a number of tokens and waits until those tokens are allocated. When the tokens are allocated, the job proceeds to another node according to the routing specification for the allocate node. Though the job is moving among other nodes, it is considered to remain at the queue and be among the jobs at the particular allocate node until it frees or destroys the tokens it holds. A job already holding tokens at a passive queue may not visit an allocate node of that queue until it releases or destroys those tokens. Jobs holding tokens are considered part of queue length performance measures for the passive queue and allocate node, and the time spent holding tokens is included in the queueing time measures for the passive queue and allocate node. Where passive queues are used for measuring response times, response times may be placed in categories at the beginning of the response time by choice of allocate node. (See Section 5.6 for discussion of categorization of mean response times at the end of a response time.) A "job copy" remains in the data structure for the passive queue and allocate node; the job receiving tokens has a pointer to this job copy.

The declaration of allocate nodes consists of one or more pairs or triples of lines containing allocate node lists and allocate node characteristics, depending on the queueing discipline. There will be pairs of lines for FCFS and FF and triples of lines for PRTY. The first element of a pair will be "ALLOCATE NODE LIST:" followed by a list of allocate nodes, analogous to the class list lines of active queues. The second element of a pair will be a list of expressions indicating the numbers of tokens requested by jobs at allocate nodes in the allocate node list. This line consists of "NUMBERS OF TOKENS TO ALLOCATE:" followed by a list of

expressions and is similar to the service times and work demands lines of active queues. The differences are the change in keywords and that the expressions are always used "as is" and never interpreted as the mean of an exponential distribution. If the result of the expression is not an integer, the result is rounded to the nearest integer. In the case of the PRTY queueing discipline, the allocate node list and numbers of tokens to allocate lines are followed by a list of priority expressions, as with active queues with the PRTY discipline.

The following example illustrates definition of allocate nodes:

```
ALLOCATE NODE LIST:                a    b
   NUMBERS OF TOKENS TO ALLOCATE:1    discrete(10,.5;20,.5)
   PRIORITIES:                     5    jv(j_prty)
ALLOCATE NODE LIST:                c d e(*)  /*"(*)" is optional*/
   NUMBERS OF TOKENS TO ALLOCATE:user(jv(leng)+3;4.4)
   PRIORITIES:                     3
ALLOCATE NODE LIST:                f(*)
   NUMBERS OF TOKENS TO ALLOCATE:g(*)
   PRIORITIES:                     h(*)
```

5.2. AND Allocate Nodes

An AND allocate node is similar to a plain allocate node, but may be associated with more than one passive queue. A job arriving at an AND allocate node requests a (possibly different) number of tokens from each of the queues associated with the node and waits until all of those tokens can be allocated simultaneously. None of the requested allocations are made until all can be made. After the allocations are made, it is as if the job sequentially visited "plain" allocate nodes of each of the queues except that the allocation was performed at a single node and that the allocations occurred simultaneously. Release or destruction of the tokens occurs as if the tokens had been acquired at "plain" allocate nodes. Separate performance measures are maintained for the node for each of the queues with which it is associated.

Different queueing disciplines may be specified for the several passive queues associated with an AND allocate node. The simultaneous allocation requirement imposes an additional constraint on scheduling. FCFS and PRTY do not allow allocation out of order when allocation in order is not currently possible. FF does allow scheduling out of order when simultaneous allocation requirements prevent allocation in order. For example, let us suppose that an AND allocate node is associated with queues A and B. With FCFS or PRTY scheduling at both queues, if the next job in line at queue A is at an AND allocate node and there are sufficient tokens to satisfy that job's request at queue A but the job is not next in line at queue B or there are insufficient tokens to satisfy its request at queue B, then the job must wait, and any jobs behind it in line at queue A must also wait, even if they are not at AND allocate nodes. With FCFS or PRTY scheduling at queue A and FF at queue B, if the next job in line at queue A is at an AND allocate node and there are sufficient tokens to satisfy that job's requests at both queues, then even if the job is not next in line at queue B it will receive the requested tokens at both queues (assuming the jobs ahead of it in line at queue B can not receive their requested tokens).

AND allocate nodes may be declared only in dialogue files. The declaration of AND allocate nodes follows the same rules as for "plain" allocate nodes except that the node list line consists of "AND ALLOCATE NODE LIST:" followed by a list of AND allocate nodes. These nodes will be declared with each queue with which they are to be associated, and may have different token requirements and priorities in these different declarations.

The following example illustrates definition of AND allocate nodes:

```
AND ALLOCATE NODE LIST:              a    b
   NUMBERS OF TOKENS TO ALLOCATE:1    discrete(10,.5;20,.5)
   PRIORITIES:                       5    jv(j_prty)
AND ALLOCATE NODE LIST:              c d e(*)  /*"(*)" is optional*/
   NUMBERS OF TOKENS TO ALLOCATE:user(jv(leng)+3;4.4)
   PRIORITIES:                       3
AND ALLOCATE NODE LIST:              f(*)
   NUMBERS OF TOKENS TO ALLOCATE:g(*)
   PRIORITIES:                       h(*)
```

5.3. OR Allocate Nodes

An OR allocate node is similar to a plain allocate node, but may be associated with more than one passive queue. A job arriving at an OR allocate node requests a (possibly different) number of tokens from each of the queues associated with the node and waits until one of those requests can be satisfied. None of the other requested allocations are made. If several of the requests can be satisfied, then the first queue in the dialogue which can satisfy a request for the OR allocate node is the queue chosen to satisfy the request. After the allocation is made, it is as if the job visited only a "plain" allocate node of the queue which satisfied its request except that the waiting times for the unsatisfied requests are treated as queueing times at the corresponding queues. Release or destruction of the tokens occurs as if the tokens had been acquired at "plain" allocate nodes. Separate performance measures are maintained for the node for each of the queues with which it is associated.

Different queueing disciplines may be specified for the several passive queues associated with an OR allocate node. OR allocate nodes may be declared only in dialogue files. The declaration of OR allocate nodes follows the same rules as for "plain" allocate nodes except that the node list line consists of "OR ALLOCATE NODE LIST:" followed by a list of OR allocate nodes. These nodes will be declared with each queue with which they are to be associated, and may have different token requirements and priorities in these different declarations.

The following example illustrates definition of OR allocate nodes:

```
OR ALLOCATE NODE LIST:               a    b
   NUMBERS OF TOKENS TO ALLOCATE:1    discrete(10,.5;20,.5)
   PRIORITIES:                       5    jv(j_prty)
OR ALLOCATE NODE LIST:               c d e(*)  /*"(*)" is optional*/
   NUMBERS OF TOKENS TO ALLOCATE:user(jv(leng)+3;4.4)
   PRIORITIES:                       3
OR ALLOCATE NODE LIST:               f(*)
   NUMBERS OF TOKENS TO ALLOCATE:g(*)
   PRIORITIES:                       h(*)
```

5.4. Transfer Nodes

Transfers nodes are related to allocate nodes, but perform a very specialized function, transfer of tokens between related jobs. The discussion of fission nodes in Section 8 is prerequisite to this section. Transfer nodes are only for use by children. A child arriving at a

transfer node requests that either any tokens of the queue that its parent holds be transferred to the child or that any tokens of the queue that it holds be transferred to the parent. This transfer must be possible. The simulation will terminate if the transfer is not possible. A visit to a transfer node is instantaneous as far as simulated time is concerned. Transfer nodes are intended for situations where a passive queue is used for measuring response times. A response time measurement begun by a parent (child) may be terminated by a child (parent). After the token transfer occurs, it is as if the job receiving the tokens had originally made the request and been allocated the tokens, in particular, the queueing time that began when the tokens were requested continues, uninterrupted by the transfer. In the internal representation, the job copy pointer in the data structure representing the job giving up the tokens is moved to the data structure representing the job receiving the tokens.

Transfer nodes may be declared only in dialogue files. The declaration of transfer nodes follows the same rules as allocate nodes except (1) the node list line consists of "TRANSFER NODE LIST:" followed by a list of transfer nodes, (2) the numbers of tokens line consists of "NUMBERS OF TOKENS TO TRANSFER:" followed by a list of expressions and (3) there is no priorities line. The value of the number of tokens to transfer expressions will typically be either 1 or −1. If the expression is positive, then the transfer is from parent to child. If the expression is negative, then the transfer is from child to parent. Note that RESQ requires that expressions with a unary minus be parenthesized. If the magnitude of the expression is not identical to the number of tokens held by the job holding the tokens, a fatal simulation error will occur when the transfer is attempted.

The following example illustrates definition of transfer nodes:

```
TRANSFER NODE LIST:                a    b
   NUMBERS OF TOKENS TO TRANSFER:1     jv(tokns_held)
TRANSFER NODE LIST:                c d e(*)  /*"(*)" is optional*/
   NUMBERS OF TOKENS TO TRANSFER:(-1)
```

5.5. Release Nodes

Release nodes are used by a job holding tokens to return all of those tokens to the passive queue. If a job visits a release node without holding tokens of that queue, there is no effect on the job or the queue and the job proceeds according to the routing specified for the release node. A visit to a release node is instantaneous as far as simulated time is concerned. The same mechanism of token release will be performed, if necessary, by a fusion node (Section 8) or a sink (Section 9). The release of tokens ends the job's association with the queue (unless and until it makes a new token request at an allocate node). In particular, the queueing time that began when the tokens were requested ends when the tokens are released. In the internal representation, the job copy in the data structure representing the queue is returned to free storage.

The declaration of release nodes consists of one or more lines listing the names of release nodes. These lines consist of "RELEASE NODE LIST:" followed by a list of release nodes. The following example illustrates definition of release nodes:

```
RELEASE NODE LIST:a b
RELEASE NODE LIST:c d e(*)  /*"(*)" is optional*/
```

5.6. Destroy Nodes

Destroy nodes are used by a job holding tokens to destroy all of those tokens rather than return them to the passive queue. If a job visits a destroy node without holding tokens of that queue, there is no effect on the job or the queue, and the job proceeds according to the routing specfied for the destroy node. A visit to a destroy node is instantaneous as far as simulated time is concerned. The destruction of tokens ends the job's association with the queue (unless and until it makes a new token request at an allocate node). In particular, the queueing time that began when the tokens were requested ends when the tokens are destroyed. For a passive queue with both release and destruction of tokens, mean queueing time values categorized by release or destroy are available (Section 13). Thus, where passive queues are used for measuring response times, mean response times may be placed in either category, when the response time is to end, by choice of release or destroy. (If necessary, a create node may be used to add tokens to the queue to make up for the tokens destroyed.) In the internal representation, the job copy in the data structure representing the queue is returned to free storage.

The declaration of destroy nodes consists of one or more lines listing the names of destroy nodes. These lines consist of "DESTROY NODE LIST:" followed by a list of destroy nodes. The following example illustrates definition of destroy nodes:

```
DESTROY NODE LIST:a b
DESTROY NODE LIST:c d e(*)  /*"(*)" is optional*/
```

5.7. Create Nodes

Create nodes are used by a job to add new tokens to a passive queue, usually to complement the effects of a destroy node. A job visiting a create node may or may not hold tokens of that queue, the effect is the same in either case. A visit to a create node is instantaneous as far as simulated time is concerned. In representing communication protocols and similar mechanisms, it is often the case that a job will destroy tokens and later either create tokens itself or have another job create tokens. This is effectively a release of tokens, but can be used to represent delays in notification of token availability (e.g., the transmission delay for an acknowledgement).

The declaration of create nodes consists of one or more pairs of lines. The first line of the pair, listing the names of create nodes, consists of "CREATE NODE LIST:" followed by a list of create nodes. The second line of the pair consists of "NUMBERS OF TOKENS TO BE CREATED:" followed by a list of expressions for the numbers of tokens created. The following example illustrates definition of create nodes:

```
CREATE NODE LIST:                a    b
   NUMBERS OF TOKENS TO CREATE:1    discrete(10,.5;20,.5)
CREATE NODE LIST:                c d e(*)  /*"(*)" is optional*/
   NUMBERS OF TOKENS TO CREATE:user(jv(leng)+3;4.4)
CREATE NODE LIST:                f(*)
   NUMBERS OF TOKENS TO CREATE:g(*)
```

6. QUEUE TYPES

This section covers the syntax and semantics of the declaration and usage of user defined queue types. A user defined queue type is a macro definition of a queue declaration. Queue types are usually used to create several queue definitions where the differences between the definitions can be specified by parameters to the queue type. For example, if FCFS were not a predefined queue type, the user could define a queue type with the same characteristics (but somewhat different syntax).

There are two distinct operations involved in the use of queue types: the definition of a queue type and the invocation of a queue type. The queue type definition consists of the specification of a parameterized queue template in which some of the queue type characteristics are given explicit values and other queue type characteristics are left as parameters to be defined when the queue type is invoked. The explicit values become the default characteristics of the queue type. Once a queue type has been defined, it can later be invoked to create a specific declaration of a queue. A set of parameter values is given as part of the invocation. A queue declared by an invocation of a queue type assumes the default characteristics of the queue type and the parametric characteristics given by the set of parameter values in the invocation.

6.1. Definition of Queue Types

Queue type definitions are given just prior to queue definitions, in either models or submodels, as illustrated in the example of Section 1.3. A queue type definition begins with a line naming the definition, "QUEUE TYPE:" followed by the name. After the name is given there are sections, in order, for declaration of numeric parameters, distribution parameters and node parameters. The declaration of numeric parameters and distribution parameters is the same as declarations of these types of parameters at the beginning of a model or submodel, except in regard to vectors and matrices. Matrix parameters are not allowed within queue type definitions. Vectors are allowed, but the declaration of a vector does not give the number of elements in parentheses as in models and submodels. Rather, a name is declared as a vector by following the name by "(*)", with the number of elements to be determined by the value supplied for the parameter when the queue type definition is invoked. Both of these declarations are optional. Distribution parameters may only be declared in dialogue files. Node parameters have the same syntax as in submodel definitions, except that declaration of vector node parameters uses the "(*)" notation given above rather than the notation used for submodel vector parameters, but node parameters have a substantially different meaning in queue type definitions. Node parameters in queue type definitions are used to list all nodes (classes, allocate nodes, release nodes, etc.) which are to be declared within the queue type. Thus node parameter declarations are necessary in queue type definitions. In analogy to block structured programming languages such as PL/I, the names used for parameters may be names previously used for elements outside of the queue type definition. The names declared within the queue type definition are local to the queue type definition. Node parameter names cannot be reused in other queue type definitions.

After the parameter definitions, the next line gives a predefined queue type that is to be the basis of the user defined type. This line consists of "TYPE:" followed by one of the predefined general types described in Sections 4 and 5, FCFS, IS, PS, LCFS, PRTY, PRTYPR, ACTIVE or PASSIVE. After the predefined type is given, the queue type definition follows exactly the rules for that type given in Section 4 or 5, with the freedom to use numeric and distribution parameters in the expressions and the added requirement that all nodes listed have been previously declared as node parameters. The queue type definition is

terminated by a line of the form "END OF QUEUE TYPE" followed by the name of the queue type.

Following are an example queue type definition for an active queue,

```
QUEUE TYPE:q_link
   NODE PARAMETER:class_name
   TYPE:active
   DSPL:fcfs
   CLASS LIST:class_name
      WORK DEMANDS:standard(jv(0),0)
END OF QUEUE TYPE Q_LINK
```

and an example definition for a passive queue,

```
QUEUE TYPE:pfcfs              /* passive fcfs queue template */
   NUMERIC PARAMETERS:ntokens   /* number of tokens in pool */
   NODE PARAMETERS:alloc(*) releas
   TYPE:passive
   TOKENS:ntokens
   DSPL:fcfs
   ALLOCATE NODE LIST:alloc
      NUMBERS OF TOKENS TO ALLOCATE:1
   RELEASE NODE LIST:releas
END OF QUEUE TYPE PFCFS
```

6.2. Invocation of Queue Types

A queue type invocation begins as with the queue definitions discussed in Sections 4 and 5, but instead of the name of a predefined type being given on the type definition line, the name of a user defined type is given. The remainder of the queue definition supplies values (arguments) for the parameters declared in the queue type definition. There are two ways to do this, a positional short format and a format which explicitly matches parameter names and values given. The positional format is analogous to procedure calls and similar statements in programming languages. *On the type definition line,* following the name of the user defined type is a colon (":") and then a list of values, with the values separated by semicolons (";"). For example, the queue type "q__link" defined above might be used in the positional format as follows,

```
QUEUE:q
   TYPE:q_link:c
```

and the queue type "pfcfs" defined above might be used in the positional format as follows,

```
QUEUE:memory
   TYPE:pfcfs: pageframes; getmemory(*); freememory
```

In the matching format, in interactive mode there will be a prompt for every parameter, where the prompt consists of the name of the parameter followed by a colon (":") and the reply is to be the value. The prompts will be in the order the parameters were declared. In dialogue mode there must be a line for every parameter, consisting of the parameter name followed by a colon followed by the value. In dialogue mode these lines may be in any order. They need

not be in the order the parameters were declared. For example, the queue type "q_link" defined above might be used in the matching format as follows,

```
QUEUE:q
   TYPE:q_link
   CLASS_NAME:c
```

and the queue type "pfcfs" defined above might be used in the matching format as follows,

```
QUEUE:memory
   TYPE:pfcfs
   NTOKENS:pageframes
   ALLOC:getmemory(*)
   RELEAS:freememory
```

In either format, a parameter value must be either a single expression or a single name. Where parameters are declared as vectors, parameter values must also be vectors.

User defined queue types are used only for definition of queues. There should be no attempted reference elsewhere in the model definition to queue types or parameters defined within queue types. Invocation of user defined queue types is transparent to RESQ solution components, i.e., the queue definitions look the same to the solution components as they would if they had been defined using only predefined queue types.

7. SET NODES

This section covers the syntax and semantics of set nodes. Set nodes are used to perform assignment statements in the sense of programming languages. Set nodes are used to assign values to job variables, global variables and chain variables. Section 3 discusses declaration of these variables. Set nodes are represented in RESQ diagrams by rectangles showing the assignment statements performed.

The declaration of set nodes, if any are to be declared, follows immediately after the queue definitions section in either a model or submodel. Set node declarations consist of pairs of lines, the first line giving a list of set nodes and the second line giving a list of assignment statements. The set node list line consists of "SET NODES:" followed by one or more names of set nodes. The names may be names of node arrays. An entire node array is indicated either by just the name or by the name followed by "(*)".

A set node assignment consists of the variable to be assigned, followed by an equals sign ("="), followed by the expression to be evaluated and assigned to the variable. The variable to be assigned must be a single variable, i.e., a single assignment may not be used to assign values to more than one element of a vector or matrix. If the variable to be assigned is an element of a vector or matrix, the subscript expressions may be simulation dependent. The subscript expressions, if any, are evaluated before the expression to be assigned is evaluated. The expression to be assigned may be simulation dependent.

The assignment list line consists of "ASSIGNMENT LIST:" followed by one or more assignments. If the node list line lists exactly one name (perhaps the name of a node array, indicating the entire array), then the assignment list line lists one or more assignments to be performed at that set node. (The list applies to each element of a node array if a node array name is given.) These assignments are performed in the order listed when a job visits the set node. If the node list line lists more than one name, then only one assignment may be performed at each set node in that list. The assignment list line must list the same number of assignments as the node list line lists names.

The following example illustrates the declaration of set nodes:

```
SET NODES:a
ASSIGNMENT LIST:jv(msg_origin)=1                              ++
               orig_count(1)=orig_count(1)+1                  ++
               jv(msg_dest)=discrete(2,1/3;3,1/3;4,1/3)  ++
               jv(msg_lng)=uniform(40,1000,1)
SET NODES:b c(*)
ASSIGNMENT LIST:alpha=beta+discrete(1.3,.5;10,.5)  jv(0)=jv(0)+1
SET NODES:set_d_cw
ASSIGNMENT LIST:delay_cw(jv(msg_origin);jv(msg_dest))=          ++
               alpha*(clock-jv(msg_atime))                      ++
               +(1-alpha)*delay_cw(jv(msg_origin);jv(msg_dest))
```

8. SPLIT, FISSION, FUSION AND DUMMY NODES

This section covers the syntax and semantics of the declaration and usage of split, fission, fusion and dummy nodes. Figure 8.1 shows the diagram RESQ diagram symbols for these nodes. Full understanding of this section presumes knowledge of Section 9 (Routing Chains), but this section is intended to be readable prior to reading Section 9.

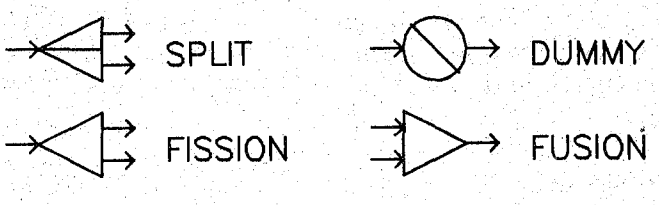

Figure 8.1 - Split, Fission, Fusion and Dummy Nodes

8.1. Split Nodes

Split nodes allow a job to produce additional independent jobs. Split nodes are useful in representing bulk arrival mechanisms and in representing control messages (e.g., acknowledgements) in communication system protocols. The third example in Appendix 1 illustrates this latter application. A split node has one entrance, an exit for the job that entered and an additional exit for each new job to be produced. The newly produced jobs are given the same job variable values as the existing job. The newly produced jobs do not possess tokens, whether or not the existing job possessed tokens. A visit to a split node is instantaneous, as far as simulated time is concerned.

The routing syntax implicitly declares names of split nodes. It is not necessary to give the name of a split node before the routing definition. However, names of split nodes may be explicitly declared prior to the routing definition. Such declarations may help clarify a model definition and prevent errors. Explicit declarations of split nodes may be given only in dialogue files. Declarations of split nodes, if any are to be made, are next in sequence following definition of set nodes. Split node declarations consist of one or more lines consisting of "SPLIT NODES:" followed by a list of split nodes, e.g.,

```
SPLIT NODES:a b c d(*)
SPLIT NODES:e
```

The routing to a split node is defined as with other nodes, e.g., if "y" is the name of a split node, we might have

```
:x->y z;.9 .1
```

If a name of a split node is not declared prior to the implicit declaration in the routing, a warning message will occur at the point of implicit declaration, e.g.,

```
**ERROR**  WNG: THE NODE "Y          " HAS BEEN IMPLICITLY DECLARED
```

At this point in the example, assuming no previous declaration, it is only known that "y" is the name of a previously undeclared node. Y might be the name of a fission or dummy node instead of a split node. The routing from a split node consists of the name of the split node, followed by an arrow ("->"), followed by a list of at least two names of nodes (not necessarily distinct names), followed by ";split". For example, we might have

```
:y->alpha alpha beta;split
```

If the name of the split node has not previously been declared as a split node, it is a routing line of this form that indicates the name is the name of a split node. The number of jobs to be produced is one less than the number of nodes in the list of nodes. The first node in the list of nodes is the destination for the existing job (the one that enters the split node). The remaining nodes in the list of nodes are the destinations for the newly produced jobs. In the above example there would be two newly produced jobs. The existing job and one of the new jobs would go to node alpha and the other new job would go to node beta.

8.2. Fission and Fusion Nodes

Fission nodes allow a job to produce additional jobs dependent on the existing job. Fusion nodes allow for the destruction of the newly produced jobs in a coordinated manner. Fission and fusion nodes are usually used together in pairs. Fission and fusion nodes are useful for representing synchronized processes (tasks) occurring in operating systems. Similarly, fission and fusion nodes are useful for representing parallel physical activities representing a single logical activity, for example transmission of a message across a communication network as a collection of packets.

A fission node has one entrance, an exit for the existing job (referred to as the "parent"), and an additional exit for each new job to be produced. The produced jobs are referred to as "children." Children may themselves enter fission nodes, thus producing hierarchies of jobs. Children are given the same job variable values as the parent. The children do not possess tokens, whether or not the parent does. A visit to a fission node is instantaneous, as far as simulated time is concerned. *Jobs are not allowed to leave the network (i.e., by going to sinks) as long as they have relatives (parents or children).* If this rule is violated, the simulation terminates.

In RESQ diagrams a fission node is represented by a triangle with the entrance at one vertex and the exits on the opposite side. This corresponds to the split node representation except that the triangle is not divided into separate sub-triangles for the parent and child exits. In the dialogue syntax, fission nodes are treated exactly the same as split nodes, except that (1) the keyword "FISSION" is used instead of the keyword "SPLIT," (2) there is an interactive prompt to optionally declare the names of fission nodes, and (3) in dialogue files, if the names of fission nodes are declared before the routing definition they are declared after declarations for split nodes, if any split node declarations are present.

A fusion node provides a place for jobs to wait for related jobs (parents or children). A fusion node has no effect on jobs without relatives. Such jobs pass through a fusion node without delay or other effect. No more than one job of a "family" can stay at a fusion node. If a job arrives at a fusion node and it has relatives, but none of its relatives are at this particular fusion node, it waits at the fusion nodes. When a job arrives at fusion node and it has a relative at this particular fusion node, two things can happen, depending on the relationship between the jobs. If one is the parent and the other is a child, then the offspring is destroyed. If both are children, the one that was produced last is destroyed. Before a child is destroyed, any tokens it holds are released. After destruction of one job, if the other job has

no remaining relatives, it proceeds from the exit of the fusion node. If the other job still has other relatives, it waits at the fusion node for another relative to arrive.

In RESQ diagrams fusion nodes are represented by a triangle with the exit(s) at one vertex and the entrance(s) on the opposite side. Names of fusion nodes must be declared as such. The declarations follow the declarations of fission nodes, if any fission nodes are declared. A fusion node declaration line consists of "FUSION NODES:" followed by a list of names of fusion nodes, e.g.,

```
FUSION NODES:a
FUSION NODES:b c d(*) e
```

As mentioned above, a child may go to a fission node to produce its own children. There are two rules which must be kept in mind:

1. Whenever a job visits a fission node, it produces its immediate descendents, i.e., a job can never directly produce grandchildren.
2. Related jobs more than one generation apart, e.g., grandparents and grandchildren, may not be present at the same fusion node. If this rule is violated, the simulation will terminate.

An immediate consequence of these rules is that it is usually necessary to have (at least one) separate pair of fission and fusion nodes for every generation of jobs that is to be produced.

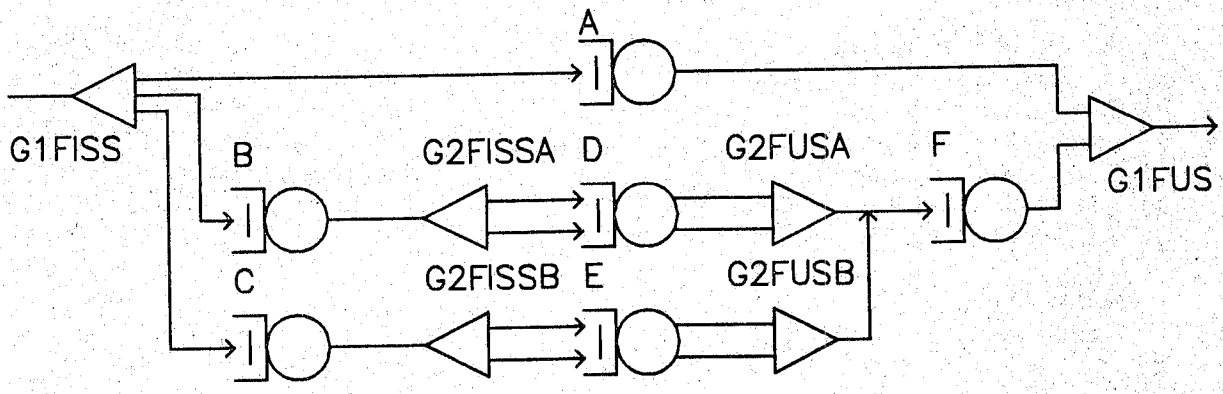

Figure 8.2 - Nesting of Fission and Fusion Nodes

Figure 8.2 illustrates an abstract set of fission and fusion nodes which might be tailored to a variety of purposes. For example, suppose a communication network is such that messages must be broken into packets for transmission and must be broken into sub-packets for transmission across certain links. Further, a message consists of exactly two packets and a packet consists of exactly two sub-packets. Node g1fiss (generation 1 fission) in the figure could represent breaking the message into packets. Since a job that enters g1fiss cannot directly generate grandchildren, it generates two children, representing the packets. Queue a would be eliminated in this case and the jobs that enter g1fiss would go directly to g1fuse. The children leaving g1fiss would be transmitted across the portion of the network allowing

full packets, e.g., queues b and c in the figure. Then they reach g2fissa and g2fissb, where they produce children to represent breaking the packets into sub-packets. A child represents one sub-packet and a grandchild represents the other. After transmission across the portion of the network requiring sub-packets, e.g., queues d and e in the figure, a child and grandchild can reunite at the generation 2 fusion nodes to represent assembling the sub-packets into packets. The child (packet) then proceeds further across the network, e.g., through queue f in the figure to the generation 1 fusion node. When both children have reached the fusion node, their parent (representing the reassembled message) leaves the fusion node.

Many other situations can be represented by tailoring of the figure. In some situations it would be appropriate to eliminate the second child and its grandchild (the ones associated with queue c, g2fissb, queue e and g2fuseb). Note that it would not be correct to have another fission/fusion pair along the parents path. In that case, the parent would stay indefinitely at the added fusion node after arriving at that fusion node, while the children produced at g1fiss would stay indefinitely at g1fuse after arriving at g1fuse.

8.3. Dummy Nodes

Dummy nodes are used in routing definitions to allow specification of routing not otherwise possible and/or to clarify specification of routing. Dummy nodes have no other effect on the jobs or the network. With split and fission nodes, the syntax of the routing does not allow decision mechanisms (probabilities and/or predicates) for jobs leaving the exits. The exits may be names of dummy nodes, and then the normal decision mechanisms may be used with regard to routing from the dummy nodes. With submodels, only one primary entry point (the input synonym) and one primary exit point (the output synonym) may be defined per external routing chain. A dummy node may be used as the primary entry (exit) point when more than one entry (exit) point is desired. However, the use of node parameters should be considered as an alternative in this situation. The second example in Appendix 1 illustrates both approaches.

Names of dummy nodes may be implicitly declared as with names of split and fission nodes. In the case of dummy nodes, there will be nothing in the routing explicitly identifying the node as a dummy node. The same warning message discussed with split nodes will occur with implicit declaration of dummy node names. Dummy node names may be declared explicitly as with split and fission nodes. Dummy node declarations may be given only in dialogue files. Dummy node declarations follow fusion node declarations. Each line consists of "DUMMY NODES:" followed by a list of dummy nodes, e.g.,

```
DUMMY NODES:a
DUMMY NODES:b c d(*) e
```

9. ROUTING CHAINS

This section covers the syntax and semantics of the declaration of routing chains. Routing chains define the routing among nodes of the network, i.e., they chain the nodes together. Routing chains are usually referred to simply as "chains." Each node of the network belongs to exactly one chain, with the exception of the predefined node "sink" which may be used in several chains.

There are two basic types of chains, closed and open. Closed chains have a fixed number of jobs (the "population") which remain among the nodes of the chain throughout the simulation. Open chains have a (usually) fluctuating number of jobs. Jobs leave the chain (and the network, simultaneously) by going to the predefined node "sink", which may be used in all open chain definitions. An open chain usually also has one or more sources for external arrival of jobs, but sources are not strictly necessary in an open chain since jobs initially placed in the chain may produce additional jobs by visiting split nodes. Initial placement of jobs at the beginning of simulation, for both closed and open chains, is discussed in Section 12. Sources are declared within open chain declarations. Figure 9.1 shows the diagram symbols for sources and sinks. (The symbols are the same except for direction of the arrows.)

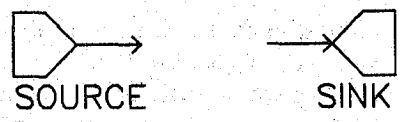

Figure 9.1 - Source and Sink

In addition to basic chain types, closed and open, submodels have chains declared as "external." External chains are those declared as chain parameters at the beginning of a submodel definition (Section 3). An external chain in a submodel is really only part of a chain, with the remaining part to be defined in the model (or submodel) that invokes the submodel. An external chain is determined to be either closed or open by the type of chain that it is connected to. Submodels may also have chains which are strictly internal to the submodel; these chains are declared as closed or open in the submodel.

The definitions of chains in the model proper and of internal chains in submodels follow the same rules. The definition of external chains is sufficiently similar that we discuss definition of routing chains in general and indicate the differences between these two situations as appropriate.

The definition of routing chains within a model or submodel follows the definition of queues, other nodes (set nodes, split nodes, fission nodes, fusion nodes, dummy nodes), submodels and submodel invocations so that all nodes to appear in the routing have been declared. Submodels and submodel invocations are discussed in Section 10. (Sources are necessarily associated with a particular chain and are declared within the chain definition. Split, fission and dummy nodes may be implicitly declared in the routing as discussed in Section 8.) Chains which are not elements of chain arrays are declared individually. Chain arrays are declared collectively. We first consider chains which are not elements of chain arrays, then we consider chain arrays.

9.1. Individual Chain Definitions

A chain definition begins with a line with "CHAIN:" followed by the name of the chain. This will be the first occurrence of the name of the chain unless (1) this is an external chain

definition within a submodel, in which case the name will have previously been declared as a chain parameter or (2) this chain name has been supplied as the value for a chain parameter in an invocation of a submodel. Following the chain name line is the chain type line, which consists of "TYPE:" followed by "closed", "open" or "external". The next few lines, preceding the actual definition of routing within the chain, are dependent on the chain type.

9.1.1. Closed Chain Definitions

With a closed chain, the only line preceding the routing definition lines is a line giving the chain population. This line consists of "POPULATION:" followed by an expression for the number of jobs in the chain. This expression must be simulation independent (as defined in Appendix 3). For example, we might have

```
CHAIN:c
   TYPE:closed
   POPULATION:users
```

for the beginning of the declaration of a closed chain.

9.1.2. Open Chain Definitions

With an open chain there will usually be a pair of lines preceding the actual routing definitions, for declaration of sources. Declaration of sources is optional, because of the possible use of split nodes suggested above and the possible definition of sources with external chain definitions as discussed below, but if no sources are declared in an open chain, a warning message will be produced. There is only provision for one pair of source declaration lines because a single source is usually sufficient, and if many sources are necessary, concatenation may be used to make each line of the pair arbitrarily long. The first of the pair lists the names of the sources and consists of "SOURCE LIST:" followed by the names. The second line defines the interarrival time distributions for these sources. It consists of "ARRIVAL TIMES:" followed by a list of expressions, one per source. For example, we might have

```
CHAIN:c
   TYPE:open
   SOURCE LIST:s
   ARRIVAL TIMES:1/msg_rate
```

for the beginning of the declaration of an open chain.

If an arrival time expression, after resolution of parameters and identifiers, contains no references to RESQ probability distribution keywords, then the value of the expression is interpreted as the mean of a (negative) exponential probability distribution. (Exponential interarrival times produce a Poisson arrival process.) (The RESQ probability distribution keywords are BE, DISCRETE, STANDARD and UNIFORM. See Appendix 3 for further discussion of distributions.) If the expression does contain at least one distribution keyword, then the expression is used directly. In either case, when a source arrival is to be scheduled, a sample is obtained from the arrival time distribution. If CV(0) is 1 for this chain (as it is initially), then the next arrival is scheduled at the current time plus the arrival time sample. However, if CV(0) for the chain varies from one, the timing mechanism is more complex. Changing CV(0) (using a set node) gives arrival times dependent on the current state of the simulation. This can be used to give arrival times dependent on simulated time (for example, to represent arrival processes dependent on time of day), on numbers of jobs at various queues (to represent arrivals dependent on congestion), etc. CV(0) is used as an arrival *rate* factor. Assuming CV(0) is positive, all samples from arrival time distributions are divided by CV(0)

to give the time until the next arrival. If CV(0) changes between the time the arrival is scheduled and the scheduled time of the arrival, then the remaining time until the arrival is multiplied by the old value of CV(0) and (assuming CV(0) is still positive) divided by the new value of CV(0). The arrival is rescheduled at the current time plus this modified remaining time. If CV(0) ever becomes zero (or negative) then the source is shut off and will produce no more arrivals during the simulation, regardless of future changes to CV(0).

9.1.3. External Chain Definitions

With an external chain, prior to the actual routing definition there is a pair of lines to define the input and output synonyms. The first line of the pair consists of "INPUT:" followed by the name of a single node in the submodel which may be referred to as "input" in the invoking model (or submodel). The second line of the pair consists of "OUTPUT:" followed by the name of a single node in the submodel which may be referred to as "output" in the invoking model (or submodel). In dialogue files only, a second pair of lines may be given to define sources to be part of the chain. The rules for this pair is the same as for the source declaration pair of lines in open chains. Source declarations here force the value given to the chain parameter being defined as an external chain to be an open chain. Following is an example of a possible definition

```
CHAIN:c
   TYPE:external
   INPUT:getmemory
   OUTPUT:freememory
```

for the beginning of the declaration of an external chain.

9.1.4. Routing Definitions

Following the chain type specific declarations discussed in Sections 9.1.1-9.1.3, the remainder of the chain definition is a series of lines defining the routing among the nodes of the chain. These lines are optional in an external chain, as illustrated in the chain definition in submodel "iosys" in the example in Section 1.3. Each of these lines begins with a colon (":") and describes the routing between two or more nodes.

The simplest routing line declares an unconditional directed path between two nodes. It consists of a colon (":") followed by a node name, followed by an arrow ("->"), followed by another node name. For example,

```
:a->b
```

declares that jobs leaving node A always go to node B. Lines of this form describing a sequence of nodes may be concatenated, e.g., the lines

```
:a->b
:b->c
```

may be replaced equivalently by

```
:a->b->c
```

The node names in lines of these forms, and all of the other forms we discuss in this section, may be individual elements of node arrays, e.g.,

```
:d(3)->e(primarysys+1)
```

Expressions indicating individual elements must be simulation independent (see Appendix 3). The node names in lines of these forms, and all of the other forms we discuss in this section, may be submodel input/output synonyms, qualified by the submodel invocation name, e.g.,

```
:invoc1.output->invoc2.input
:invoc2.output->invoc1.input
```

Several separate unconditional paths may be grouped together, e.g.,

```
:a->b
:node1->node2
:node3->node4
```

may be expressed on a single line as

```
:a node1 node3->b node2 node4
```

In cases like these where the nodes on the right side are the same, the node name need not be repeated, and additional paths may be added on the right. For example, the lines

```
:disk1->cpu
:disk2->cpu
:disk3->cpu
:cpu->drum
```

may be expressed on a single line as

```
:disk1 disk2 disk3->cpu->drum
```

Paths must not be specified more than once, e.g., the following would be incorrect:

```
:disk1->cpu->drum
:disk2->cpu->drum
:disk3->cpu->drum
```

(This example would produce an error message from the EVAL command that the probabilities from node "cpu" do not sum to 1.) A set of unconditional paths between node or invocation arrays may also be expressed on a single line, provided both arrays involved have the same number of elements. For example, if A and B are invocation arrays, each with N elements, then the set of lines

```
:a(1).output->b(1).input
:a(2).output->b(2).input
 ...
:a(N).output->b(N).input
```

may be expressed on a single line as

```
:a(*).output->b(*).input /*"(*)" is optional*/
```

Conditional routing may be based either on probabilities or on "predicates." A predicate is an expression with a true or false value. The simplest conditional routing line begins as with

a simple unconditional path, i.e., a colon (":") followed by a node name, followed by an arrow ("->"), followed by another node name. A semicolon (";") and either a probability expression or a predicate expression follows the second node name. For example, we might have

```
:a->b;pb
:a->c;pc
:a->d;1-(pb+pc)
```

or

```
:x->y;if(jv(count)>0)
:x->z;if(t)
```

Probability expressions must have values in the [0,1] interval. (An unconditional path is represented internally as a conditional path with probability 1.) Probability expressions may be simulation dependent (Appendix 3). Predicate expressions are normally simulation dependent; otherwise the routing may be expressed unconditionally. Predicate expressions begin with "if(" and end with ")". Conditional expressions are evaluated in the order listed, e.g., in the above example the predicate for the path from X to Y will be evaluated first. "T" represents the constant "true" value, e.g., in the above pair of lines, jobs leaving node X will always go to node Z if they do not go to node Y. Predicates are defined in detail in Appendix 3.

Predicates and probabilities may be mixed in describing conditional routing. For example, if we want to go in a clockwise direction if recent delays have been shortest in that direction and in a counterclockwise direction if delays have been shortest in that direction but, if recent delays have been the same in each direction, choose randomly between the two directions, we might have

```
:source1->cw_path;if(delay_cw<delay_ccw)
:source1->ccw_path;if(delay_ccw<delay_cw)
:source1->cw_path;.5
:source1->ccw_path;.5
```

The possible destinations are considered in order. Predicates are evaluated independently of probabilities. Probabilities are evaluated as if predicates were not involved. The following algorithm defines the mechanism more formally:

```
next_node_chosen=false
random_value=uniform random number on (0,1) interval
do while(¬next_node_chosen)
   if list_of_destinations is empty then
      signal error('no destination found')
   get next possible destination
   if probability for this destination then
      if random_value<probability then
         next_node_chosen=true
      else
         random_value=random_value-probability
   else /*predicate*/
      next_node_chosen=predicate
end
```

When the algorithm terminates normally, the last destination examined is the one used.

Several conditional paths may be grouped together on the same line when the node left is the same. For example, the last three examples could be expressed on the same line as

```
:a->b c d;pb pc 1-(pb+pc)
:x->y z;if(jv(count)>0) if(t)
:source1->cw_path ccw_path cw_path ccw_path; ++
 if(delay_cw<delay_ccw) if(delay_ccw<delay_cw) .5 .5
```

Conditional paths may be added on the right side of lines with unconditional paths, e.g.,

```
:u->v->w->x->y z;if(jv(count)>0) if(t)
```

Where a line expresses all conditional paths, those paths have equal probabilities and the probabilities are the inverse of the number of nodes explicitly named, the semicolon and probabilities may be omitted, e.g.,

```
:a->b c d;1/3 1/3 1/3
```

may be expressed as

```
:a->b c d
```

However, if E is a node array with 2 elements,

```
:a->b c d e(*)
```

would be equivalent to

```
:a->b c d e(1) e(2);1/4 1/4 1/4 1/4 1/4
```

an incorrect specification because the probabilities do not sum to one. The EVAL command or its equivalent would detect this, but the SETUP command would not, since the expressions might depend on numeric parameters.

9.2. Chain Array Definitions

All elements of a chain array are defined collectively and may not be defined individually. All references to nodes, except in status functions in predicates, must be to node arrays with the same numbers of elements as the chain array. Numeric values may be given by scalar expressions, which will be interpreted as vectors with homogeneous elements, by numeric vectors with the same numbers of elements as the chain array, or, in situations involving invocation arrays, by numeric matrices where the numbers of rows are the same as the numbers of elements in the chain array and the numbers of columns are the same as the numbers of elements in the invocation arrays. Distribution values are given either by scalar expressions, which are interpreted as vectors with homogeneous elements, or by distribution vectors. Predicates may be given only as scalar expressions, which are interpreted as vectors with homogeneous elements. Names of vectors are optionally followed by "(*)" and names of matrices are optionally followed by "(*;*)".

The form of chain array definitions is essentially the same as that of individual chain definitions. The definition begins with the "CHAIN:" line. The name of the (previously declared) chain array is given. The chain types allowed are the same as with individual chains. With closed chains, the population line gives a numeric array with the respective populations

or a numeric expression giving the population to be used for all of the elements of the chain array. With open chains, the names given for the source list must be (previously declared) names of node arrays. The elements in the arrival time list will either be names of arrays (numeric or distribution) or scalar expressions to be interpreted as homogeneous vectors. With external chains, the input/output synonyms will be (previously declared) node arrays. The source definitions, if given, will follow the same rules as with open chains. The routing definitions follow essentially the same rules as with individual chains, with node arrays taking the place of individual nodes.

The following example illustrates definition of a chain array.

```
CHAIN:interactiv(*)
   TYPE:external
   INPUT:setcmdtype(*)
   OUTPUT:freememory(*)
   :setcmdtype(*)->getmemory(*)->cpu(*)->iosys(*).input(*);prob(*;*)
   :iosys(*).output(*)->decrcycles(*)
   :decrcycles(*)->cpu(*) freememory(*);if(jv(cyclecount)>0) if(t)
```

The main point to be noticed is the mapping of the rows and columns of the matrix "prob." Prob(*;1) contains the probabilities of the conditional paths from cpu(*) to iosys(1).input(*), prob(*;2) contains the probabilities of the conditional paths from cpu(*) to iosys(2).input(*), and so on. This definition would have the same interpretation if just the array names were given, without the "(*)" and "(*;*)".

10. SUBMODELS

This section covers the syntax and semantics of the declaration and invocation of submodels. Previous sections have covered most of the components of submodel declaration, since these components are essentially the same as the components of model definitions. This section will give a global look at submodel declaration and a detailed look at invocation of submodels. The examples in Appendix 1 illustrate some of the issues discussed here.

10.1. Submodel Declarations

Submodel declarations follow the declarations of all queues and nodes in the enclosing model or submodel, e.g., after declaration of dummy nodes. A submodel declaration begins with a line declaring the name of the submodel, "SUBMODEL:" followed by the name of the submodel. The sections of a submodel declaration parallel the sections of model definitions. In order, they are

- Declaration of parameters, identifiers, variables and arrays (Section 3). At least one chain parameter must be declared. Otherwise this section is optional.
- Declaration of queue types (Section 6). This section is optional.
- Declaration of queues (Sections 4 and 5). At least one queue or node must be declared within a submodel. Otherwise this section is optional.
- Declaration of set nodes (Section 7). At least one queue or node must be declared within a submodel. Otherwise this section is optional.
- Declaration of split, fission, fusion and dummy nodes (Section 8). At least one queue or node must be declared within a submodel. Otherwise this section is optional.
- Declaration of submodels. Submodel declarations may be nested within submodels, as illustrated in the example of Section 1.3. This section is optional.
- Invocations of submodels (Section 10.2). Submodel invocations may be nested within submodels, as illustrated in the example of Section 1.3. This section is optional.
- Declaration of routing chains (Section 9). At least one external chain must be declared within a submodel. Otherwise this section is optional.

The end of a submodel declaration is indicated by a line of the form "END OF SUBMODEL name" where "name" is the submodel name.

As in nested procedure definitions in block structured programming languages (e.g., PL/I or Pascal), names used outside of a submodel definition may be reused within submodel definitions. When names are reused in this manner, the new definition persists within the submodel definition and the old definition is restored after the submodel definition is completed.

Following is the submodel definition used in the example of Section 1.3.

```
SUBMODEL:cssm /*Computer System Submodel*/
   NUMERIC PARAMETERS:pageframes
   CHAIN PARAMETERS:interactiv
   NUMERIC IDENTIFIERS:cmdtype cyclecount
      CMDTYPE:0 /*JV(0) to be used to indicate command type*/
      CYCLECOUNT:1 /*JV(1) to be used to count CPU-I/O cycles*/
   NUMERIC IDENTIFIERS:cpiocycles(3) pageneed(3)
      CPIOCYCLES: 8 15 50
      PAGENEED:  20 24 30
   NUMERIC IDENTIFIERS:cputime
      CPUTIME:.025 /*mean time in seconds*/
   QUEUE:memory
      TYPE:passive
      TOKENS:pageframes
      DSPL:fcfs
      ALLOCATE NODE LIST:getmemory
         NUMBERS OF TOKENS TO ALLOCATE:pageneed(jv(cmdtype))
      RELEASE NODE LIST:freememory
   QUEUE:cpuq
      TYPE:ps
      CLASS LIST:cpu
         SERVICE TIMES:cputime
   SET NODES:setcmdtype
      ASSIGNMENT LIST:jv(cmdtype)=discrete(1,.8;2,.15;3,.05), ++
                      jv(cyclecount)=cpiocycles(jv(cmdtype))
   SET NODES:decrcycles
      ASSIGNMENT LIST:jv(cyclecount)=jv(cyclecount)-1
   SUBMODEL:iosys
      CHAIN PARAMETERS:interactiv
      QUEUE TYPE:diskdef
         NODE PARAMETERS:servicecls
         TYPE:active
         SERVERS:1
         DSPL:fcfs
         CLASS LIST:servicecls
            WORK DEMANDS:.06
         SERVER -
      END OF QUEUE TYPE DISKDEF
      QUEUE:diskq
         TYPE:diskdef
         SERVICECLS:disk
      CHAIN:interactiv
         TYPE:external
         INPUT:disk
         OUTPUT:disk
   END OF SUBMODEL IOSYS
   INVOCATION:iosys1
      TYPE:iosys
      INTERACTIV:interactiv
   INVOCATION:iosys2
      TYPE:iosys: interactiv
   CHAIN:interactiv
      TYPE:external
      INPUT:setcmdtype
```

```
        OUTPUT:freememory
        :setcmdtype->getmemory->cpu->iosys1.input iosys2.input;.5 .5
        :iosys1.output iosys2.output->decrcycles
        :decrcycles->cpu freememory;if(jv(cyclecount)>0) if(t)
END OF SUBMODEL CSSM
```

10.2. Submodel Invocations

Invocation of a submodel creates an actual subnetwork with the characteristics of the submodel declaration. The remaining characteristics of the subnetwork created by the invocation are specified by the parameters given with the invocation. The queues, nodes and global variables defined in the submodel declaration do not actually exist until the submodel is invoked. The queues nodes and global variables are properly part of the invocation and not the submodel.

A submodel invocation begins with the line naming the invocation, "INVOCATION:" followed by the name of the invocation. The remainder of the invocation is syntactically the same as the invocation of queue types discussed in Section 6. The second line begins with "TYPE:" followed by the name of the submodel to be invoked. The remainder of the invocation supplies values (arguments) for the parameters declared in the submodel definition. There are two ways to do this, a positional short format and a format which explicitly matches parameter names and values given. The positional format is analogous to procedure calls and similar statements in programming languages. *On the type definition line,* following the name of the user defined type is a colon (":") and then a list of values, with the values separated by semicolons (";"). For example, the second invocation of "iosys" in the above example uses the positional format:

```
INVOCATION:iosys2
   TYPE:iosys: interactiv
```

In the matching format, in interactive mode there will be a prompt for every parameter, where the prompt consists of the name of the parameter followed by a colon (":") and the reply is to be the value. The prompts will be in the order the parameters were declared. In dialogue mode there must be a line for every parameter, consisting of the parameter name followed by a colon followed by the value. In dialogue mode these lines may be in any order. They need not be in the order the parameters were declared. For example, the invocation of "cssm" in the example of Section 1.3 uses the matching format:

```
INVOCATION:cssm1
   TYPE:cssm
   PAGEFRAMES:userframes
   INTERACTIV:interactiv
```

In either format, a parameter value must be either a single expression or a single name. Where parameters are declared as vectors, parameter values must also be vectors.

Invocation arrays are declared on the line naming the invocation by following the name with a parenthesized expression for the number of invocations in the array. This expression must be simulation independent (Appendix 3). The elements of invocation arrays must have the same parameter values. The two invocations of "iosys" in the example of Section 1.3 could be replaced by an invocation array with two elements, e.g.,

```
    ...
  END OF SUBMODEL IOSYS
  INVOCATION:iosys1(2)
     TYPE:iosys
     INTERACTIV:interactiv
  CHAIN:interactiv
     TYPE:external
     INPUT:setcmdtype
     OUTPUT:freememory
     :setcmdtype->getmemory->cpu->iosys1(*).input;.5
     :iosys1(*).output->decrcycles
     :decrcycles->cpu freememory;if(jv(cyclecount)>0) if(t)
END OF SUBMODEL CSSM
```

10.3. Node Parameters

In some cases a submodel may not naturally have only one entry point or one exit point for a given chain. In some cases it will be possible to add a dummy node (or nodes) to the submodel to transform it to one with a single entry point and a single exit point. In general it may not be possible or desirable to restrict a chain to having a single entry point and/or a single exit point. Node parameters may be used to provide multiple entry/exit points for a chain in a submodel.

A node parameter allows the submodel definition to refer to a node in the invoking (sub)model. *The nodes passed as parameters exist only in the invoking (sub)model. Within the (sub)model, node parameters may be used only in the routing definition and in status functions.* Thus node parameters may not be used in prompts for node lists and may not be given the input or output synonyms.

The routing definition within a submodel may specify routing directly from node parameter to node parameter. However, certain restrictions hold on the expressions allowed for routing predicates and probabilities in such a situation. We will discuss these restrictions after the following example.

Section 3 has already discussed the syntax of node parameter declarations. We now consider an abstract example to illustrate the node parameter mechanisms. The second and third examples in Appendix 1 illustrate concrete applications of node parameters.

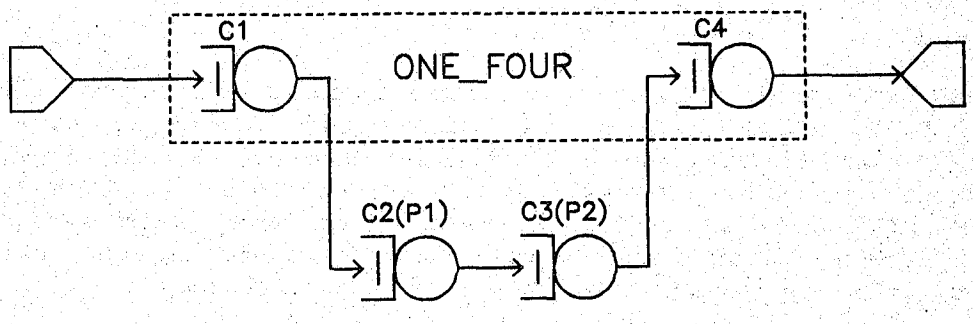

Figure 10.1 - Node Parameter Example

Suppose we wish to have a tandem network of four queues as depicted in Figure 10.1. Further, we wish to have classes c1 and c4 belong to the same submodel but have classes c2

and c3 belong to the invoking submodel. Thus we necessarily have two inputs and two outputs for the submodel. We might use the following submodel definition.

```
SUBMODEL:one_four
   NODE PARAMETERS:p1 p2
   CHAIN PARAMETERS:c
   QUEUE:q1
     TYPE:fcfs
     CLASS LIST:c1
        SERVICE TIMES:.25
   QUEUE:q4
     TYPE:fcfs
     CLASS LIST:c4
        SERVICE TIMES:.25
  CHAIN:c
     TYPE:external
     INPUT:c1
     OUTPUT:c4
     :c1->p1
     :p2->c4
END OF SUBMODEL ONE_FOUR
```

Here we have let classes c1 and c4 have the input and output synonyms, respectively, and we have let classes c2 and c3 be passed to the submodel as parameters p1 and p2. (Classes c2 and c3 are assumed to be defined in queues of the invoking model.) We could use the following invocation.

```
...
INVOCATION:inv
   TYPE:one_four
   P1:c2
   P2:c3
   C:c
CHAIN:c
   TYPE:open
   SOURCE LIST:s
   ARRIVAL TIMES:.5
   :s->inv.input
   :c2->c3
   :inv.output->sink
...
```

The definition of the routing from c2 to c3 can be expressed either in the invoking model, as we have done, or in the submodel by using a routing transition from p1 to p2. If we have a direct path specified between node parameters in a submodel definition, then the probability or predicate expression may not include references to global variables local to the submodel, may not include references to any queues (in status functions) except queues in the outermost model and may not include references to any nodes (in status functions) except nodes in the outermost model. For example, in submodel one_four a line of the form

```
:p1->p2 c4;if(ql(c2)<5) if(t)
```

would be acceptable, but

```
:p1->p2 c4;if(ql(c1)<5) if(t)
```

would not be acceptable. Note that dummy nodes may be added to a submodel to circumvent this restriction, e.g.,

```
:p1->d->p2 c4;if(ql(c1)<5) if(t)
```

where d is a dummy node (declared within the submodel) would be acceptable.

10.4. Submodel Nesting Structures

We have already discussed and illustrated common submodel nesting structures. Typically, when invocations are included within a submodel, the definition of the invoked submodel is also included in the submodel containing the invocation. However, this is not strictly necessary. Consider the following dialogue sketch.

```
MODEL:a
  ...
  SUBMODEL:b
     ...
     SUBMODEL:c
        ...
     END OF SUBMODEL C
     INVOCATION:c1
        TYPE:c
     ...
  END OF SUBMODEL B
  SUBMODEL:d
     ...
     SUBMODEL:c
        ...
     END OF SUBMODEL C
     INVOCATION:c2
        TYPE:c
     ...
  END OF SUBMODEL D
  ...
END
```

If the definition of submodel C is the same in both instances, then it would be more convenient for the user to have a single copy of the definition, so that any changes could be made once instead of twice. (It would also take less time for SETUP to process the dialogue.) Thus we might use

```
MODEL:a
  ...
  SUBMODEL:c
     ...
  END OF SUBMODEL C
  SUBMODEL:b
     ...
     INVOCATION:c1
        TYPE:c
```

```
    ...
  END OF SUBMODEL B
  SUBMODEL:d
    ...
    INVOCATION:c2
      TYPE:c
    ...
  END OF SUBMODEL D
  ...
END
```

Submodel definitions and invocations must be such that a submodel definition is in either (1) the same submodel which contains the invocation or (2) the model (i.e., it is not nested within another submodel definition). Note that these rules do not preclude having submodel definitions and invocations in submodel C. It is difficult for the simulation component of RESQ to verify that these rules have been followed; *if they are violated, the violation may not be detected.*

In situations such as this one must be careful about different elements with the same name. As in most programming languages, the "static chain of reference" is followed. The static chain of reference considers the static structure of declaration, as opposed to the "dynamic chain of reference," which considers the structure imposed by the invocations.

For example, if both the model A and the submodel B of the example have a queue named "q", and there is a reference to "q" in a status function, e.g.,

```
MODEL:a
  ...
  QUEUE:q
    ...
  SUBMODEL:c
    ...
      :alpha->beta;if(ta(q)>0)
    ...
  END OF SUBMODEL C
  SUBMODEL:b
    ...
    QUEUE:q
      ...
    INVOCATION:c1
      TYPE:c
    ...
  END OF SUBMODEL B
  SUBMODEL:d
    ...
    INVOCATION:c2
      TYPE:c
    ...
  END OF SUBMODEL D
  ...
END
```

then the two different nesting structures will give different results. In this example, the "q" referred to in the TA status function will be the one defined in the model, not the "q" defined in submodel B.

11. NUMERICAL SOLUTION

The discussion in the other sections of this document generally assumes that simulation will be used to obtain model solutions. However, numerical solution is feasible and, usually, dramatically less expensive than simulation for a subset of the models allowed by simulation. *Computational expense may be large with numerical solution with models with closed chains and substantial closed chain populations and/or with models with closed chains and several queues with queue length dependent service rates.* The first example in Appendix 1 illustrates the use of numerical solution. The numerical solution component of RESQ uses the "mean value analysis" (MVA) algorithm discussed in

S.S. Lavenberg and C.H. Sauer, "Analytical Results for Queueing Models," S.S. Lavenberg (Editor), *Computer Performance Modeling Handbook,* to appear, Academic Press (1982).
E.A. MacNair and S. Tucci, "Implementation of Mean Value Analysis for Open, Closed and Mixed Queueing Networks," to appear as an IBM Research Report.

The following restrictions apply to a model to be solved numerically:

1. In open chains, arrivals from sources must form a Poisson process. Arrival rates are constant, i.e., CV(0) must remain 1. Therefore, only an exponential interarrival time distribution can be given for each source.
2. The routing must be completely specified using only probabilities. No predicates can be used for any routing decisions.
3. The only nodes allowed are classes, sources and sinks. (Passive queues are not allowed.)
4. Only four queueing disciplines are allowed: FCFS, PS, LCFS and IS. With FCFS, there is a further restriction that all classes at a queue must have the same exponential service time distribution. No priority disciplines are permitted.
5. At queues with multiple servers all servers must have the same characteristics.
6. The performance measures produced are utilization, throughput, mean queue length and mean queueing time. No distribution estimates are available.

A model definition for numerical solution will consist of the (allowed) sections described so far, followed by a line containing an "END" after the last chain definition.

12. SIMULATION DIALOGUES

After the definition of routing chains, the definition of the model proper, i.e., the extended queueing network, is complete. However, where simulation is to be used, additional information is required (1) to indicate distributions gathered, if any, (2) to define the confidence interval estimation method, if one is to be used, (3) to dictate the initial state of the simulated system, (4) to define how the simulation run length will be determined, and (5) to define simulation tracing, if desired. The following sections discuss the syntax and semantics of the dialogue for these simulation dependent characteristics.

12.1. Distribution Gathering

By default the simulation program will gather mean performance measures and certain other measures for all queues, classes and allocate nodes (including AND and OR allocate nodes). Throughputs and departure counts are gathered by default for other nodes. Distributions of performance measures, e.g., distributions of queueing time and queue length, are only gathered upon specific user request. Gathering of distributions is less easily defined by default and may be computationally expensive unless well defined. The user may specify that distributions of queueing time and queue length be gathered for queues, classes and "plain" allocate nodes (but not AND and OR allocate nodes). In interactive mode there will be prompts for these specifications, as illustrated in Section 1.3. These specifications are optional. The user may also specify that distributions of the number of tokens in use and the total number of tokens for a passive queue be gathered. These specifications may be given only in dialogue files. We now describe these specifications in the order they would occur in dialogue files.

Cumulative queueing time distributions are gathered for queues listed on lines of the form "QUEUES FOR QUEUEING TIME DISTRIBUTION:" followed by a list of names of queues (qualified by invocation names if these queues were declared in submodels). For each queue listed there will normally be a corresponding line giving the distribution values of interest. This line consists of "VALUES:" followed by a list of expressions. These expressions must be simulation independent. The simulation program will estimate the probability that the queueing time is less than or equal to each of these values. If fewer values lines are given than the number of queues listed, the last values line will be used for the remaining queues in the list. This section for queue queueing time distributions may be repeated as necessary. For example, we might have the following specification:

```
QUEUES FOR QUEUEING TIME DIST:cssm1.memory
   VALUES:1 2 3 4 5 6 7 8
QUEUES FOR QUEUEING TIME DIST:cssm1.iosys1.diskq cssm1.iosys2.diskq
   VALUES:.03 .06 .12 .24
```

Queue length distributions will be gathered for queues listed on lines of the form "QUEUES FOR QUEUE LENGTH DISTRIBUTION:" followed by a list of names of queues (qualified by invocation names if these queues were declared in submodels). For each queue listed there will normally be a corresponding line giving the maximum queue length of interest. This line consists of "MAX VALUE:" followed by a single expression. This expression must be simulation independent. The simulation program will estimate the probability of each queue length from zero up to this maximum. If fewer of these lines are given than the number of queues listed, the last line will be used for the remaining queues in the list. This section for queue queue length distributions may be repeated as necessary. For example, we might have the following specification:

```
QUEUES FOR QUEUE LENGTH DIST:cssm1.memory
   MAX VALUE:users
QUEUES FOR QUEUE LENGTH DIST:cssm1.iosys1.diskq cssm1.iosys2.diskq
   MAX VALUE:ceil(userframes/20)
```

Token use distribution specifications are only possible in dialogue files (and only for passive queues). Token use distributions will be gathered for queues listed on lines of the form "QUEUES FOR TOKEN USE DISTRIBUTION:" followed by a list of names of queues (qualified by invocation names if these queues were declared in submodels). For each queue listed there will normally be a corresponding line giving the maximum number of tokens of interest. This line consists of "MAX VALUE:" followed by a single expression. This expression must be simulation independent. The simulation program will estimate the probability of each number of tokens in use from zero up to this maximum. If fewer of these lines are given than the number of queues listed, the last line will be used for the remaining queues in the list. This section for queue token use distributions may be repeated as necessary. For example, we might have the following specification:

```
QUEUES FOR TOKEN USE DIST:cssm1.memory
   MAX VALUE:userframes
```

Total token distribution specifications are only possible in dialogue files (and only for passive queues). Total token distributions will be gathered for queues listed on lines of the form "QUEUES FOR TOTAL TOKEN DISTRIBUTION:" followed by a list of names of queues (qualified by invocation names if these queues were declared in submodels). For each queue listed there will normally be a corresponding line giving the maximum number of tokens of interest. This line consists of "MAX VALUE:" followed by a single expression. This expression must be simulation independent. The simulation program will estimate the probability of each number of tokens in the passive queue, including tokens in use, from zero up to this maximum. If fewer of these lines are given than the number of queues listed, the last line will be used for the remaining queues in the list. This section for queue total token distributions may be repeated as necessary. For example, we might have the following specification:

```
QUEUES FOR TOTAL TOKEN DIST:windowq
   MAX VALUE:2*window-1
```

Cumulative queueing time distributions are gathered for nodes listed on lines of the form "NODES FOR QUEUEING TIME DISTRIBUTION:" followed by a list of names of nodes (qualified by invocation names if these nodes were declared in submodels). These nodes must be either classes or "plain" allocate nodes (not AND or OR allocate nodes). For each node listed there will normally be a corresponding line giving the distribution values of interest. This line consists of "VALUES:" followed by a list of expressions. These expressions must be simulation independent. The simulation program will estimate the probability that the queueing time is less than or equal to each of these values. If fewer values lines are given than the number of nodes listed, the last values line will be used for the remaining nodes in the list. This section for node queueing time distributions may be repeated as necessary. For example, we might have the following specification:

```
NODES FOR QUEUEING TIME DIST:cssm1.getmemory
   VALUES:1 2 3 4 5 6 7 8
NODES FOR QUEUEING TIME DIST:cssm1.iosys1.disk cssm1.iosys2.disk
   VALUES:.03 .06 .12 .24
```

Queue length distributions will be gathered for nodes listed on lines of the form "NODES FOR QUEUE LENGTH DISTRIBUTION:" followed by a list of names of nodes (qualified by invocation names if these nodes were declared in submodels). These nodes must be either classes or "plain" allocate nodes (not AND or OR allocate nodes). For each node listed there will normally be a corresponding line giving the maximum queue length of interest. This line consists of "MAX VALUE:" followed by a single expression. This expression must be simulation independent. The simulation program will estimate the probability of each queue length from zero up to this maximum. If fewer of these lines are given than the number of nodes listed, the last line will be used for the remaining nodes in the list. This section for node queue length distributions may be repeated as necessary. For example, we might have the following specification:

```
NODES FOR QUEUE LENGTH DIST:cssm1.getmemory
   MAX VALUE:users
NODES FOR QUEUE LENGTH DIST:cssm1.iosys1.disk cssm1.iosys2.disk
   MAX VALUE:ceil(userframes/20)
```

12.2. Confidence Intervals and Run Length

Much of the remaining dialogue depends on whether confidence interval estimation is desired, and, if so, which of three methods is chosen. An inherent problem in simulation is the statistical variability of simulation estimates of performance measures. The usual method of estimating variability of simulation results is to produce "confidence interval" estimates: given some point estimate p (e.g., for mean queueing time) and other information we estimate a confidence interval $(p - \delta, p + \delta)$. The "true" value (for the extended queueing network) is contained within the interval with some chosen probability, say .9. (The confidence interval does not indicate how accurately the extended queueing network represents the system being modeled.) This probability, expressed in percent, e.g., 90%, is known as the "confidence level." The quantity δ depends on the confidence level; the higher the confidence level is, the larger δ is. Note that the true value may lie outside of the confidence interval, but this happens only with a small probability (e.g., $1 - .9 = .1$). If a simulation is not run long enough, or if the performance measure considered is highly variable, then δ may be greater than p and $p - \delta$ may be negative even though the performance measure must be non-negative. Similarly, for performance measures known to be no greater than 1, e.g., utilizations, p and δ may be such that $p + \delta > 1$.

RESQ provides three methods for confidence interval estimation. The methods are implemented to be as transparent to the user as is practical, i.e., to minimize user decision making and to minimize required user understanding of the statistical bases of the methods. No one method is best for all applications.

- The method of independent replications is the preferred method for estimation of transient characteristics. Independent replications may be applied to estimation of equilibrium characteristics, but one of the following two methods will usually be preferable for estimating equilibrium characteristics.

- The regenerative method is the preferred method for estimation of equilibrium behavior in models with regenerative characteristics. Many models constructed with RESQ will have regenerative characteristics, but many other models will not.

- The spectral method is the preferred method for estimation of equilibrium behavior in models without regenerative characteristics. The spectral method may also be applied to models with regenerative characteristics. The regenerative method

requires more user sophistication than the spectral method in that the user must be able to define "regeneration states." Definition of a model to use the spectral method is no more difficult than definition of a model to be simulated without confidence intervals.

The regenerative method and the spectral method allow automated run length control based on achieving confidence intervals of a prespecified width. All three methods, independent replications, the regenerative method and the spectral method, are discussed from a statistical point of view in

P.D. Welch, "The Statistical Analysis of Simulation Results," S.S. Lavenberg (Editor), *Computer Performance Modeling Handbook,* to appear, Academic Press (1982).

We discuss four cases, simulation without confidence intervals and the three confidence interval methods.

12.2.1. Simulation without Confidence Intervals

After the distribution specification section, the next line is for specification of the confidence interval method. This line consists of "CONFIDENCE INTERVAL METHOD:" followed by "none", "replications", "regenerative" or "spectral". This section assumes that confidence intervals are not desired, i.e., "none" is given on the confidence interval method line.

The next major section is for specifying the initial state of the network when simulation begins, i.e., how many jobs are to be placed at which nodes. It begins with the line "INITIAL STATE DEFINITION-". Following this line there will be a triple of lines for each chain which is not empty in the initial state. Open chains with sources may be left empty in the initial state. If a model consists only of open chains with sources, then no triples need be given. A triple must be given for each closed chain or chain array. Initial states of open chains which are to be non-empty initially are specified as with closed chains. The first line of each triple identifies the chain (or chain array) and consists of "CHAIN:" followed by the name. Chain array names may optionally be followed by "(*)". The second line of each triple lists the nodes where jobs are to be placed initially. This line consists of "NODE LIST:" followed by a list of names of nodes. Where the initial state of a single chain is being defined, these must be individual nodes, i.e., elements of node arrays must be listed separately (and subscript expressions must be simulation independent). Where the initial states of chain arrays are being defined, the names of nodes in the list must be names of entire node arrays (optionally followed by "(*)"). These node arrays must all have the same numbers of elements as the chain array. The third line of the triple gives the numbers of jobs to be placed at each node in the previous list. This line consists of "INIT POP:" followed by a list of expressions and/or names of numeric vectors. For definition of initial state of a single chain, the list must consist only of simulation independent expressions, one per node listed in the node list line. For a closed chain, the sum of the values of these expressions must equal the chain population. For definition of the initial states of chain arrays, this list must have the same number of elements as the list of node names. Expressions are interpreted as values for each element in the corresponding node array. Numeric vectors must have the same numbers of elements as the chain array.

It is not possible to specify job copies in initial state definitions, e.g., it is not possible to specify that some jobs are at a class while holding tokens at an allocate node. If we want to specify jobs at a class which hold tokens, then it is necessary to place them at an allocate node in such a manner that they will immediately proceed to the desired class. For example, if in

the example of Section 1.3 we wish to have 2 jobs initially at queue "cssm1.cpuq" holding tokens of "cssm1.memory" then we should initially place those jobs at the set node "cssm1.setcmdtype":

```
INITIAL STATE DEFINITION-
CHAIN:interactiv
   NODE LIST:terminals cssm1.setcmdtype
   INIT POP: users-2   2
```

When the simulation begins, the jobs will get tokens and go immediately to "cssm1.cpuq". If we wanted to have tokens initially at a disk queue in this example, then we could add an allocate node for this purpose such that the jobs leaving the new allocate node would go directly to the disk queue and such that jobs never go to this new allocate node from other nodes. Then we could place jobs initially at this allocate node and they would go immediately to the disk queue, holding tokens, assuming sufficient tokens were available.

The next major section is for specification of simulation run length. This allows for a variety of limits to be specified. A limit on CPU time used by the simulation may also be specified after the other limits. The CPU limit is treated as a special case in some regards, especially when confidence intervals are estimated. The simulation run stops when the first of these limits is reached. (As illustrated in Section 1.3, when the run stops these limits may be increased and the run continued.)

The run limits section begins with a line "RUN LIMITS-". After that line there will lines for limits and pairs of lines for limits. These lines are all optional in dialogue files. In interactive mode, null replies to these prompts will result in "infinite" values for the corresponding limits. All of the expressions given in these lines must be simulation independent (Appendix 3). The first of these lines is for simulated time, "SIMULATED TIME:" followed by a single expression. The second of these lines is for simulated events, "EVENTS:" followed by a single expression. Simulation events are discussed in Appendix 7. Next in order are pairs of lines for limits on numbers of departures from specified queues. Several such pairs may be given, as appropriate. The first line of the pair consists of "QUEUES FOR DEPARTURE COUNTS:" followed by a list of queue names. The second line of the pair consists of "DEPARTURES:" followed by a list of expressions, one per queue listed on the previous line. Note that jobs are not counted as departures from passive queues until they release or destroy tokens, except for jobs waiting for tokens at an OR allocate node which receive tokens from some other queue of the OR allocate node. Last in order in the run limits section are pairs of lines for limits on numbers of departures from specified nodes. Several such pairs may be given, as appropriate. The first line of the pair consists of "NODES FOR DEPARTURE COUNTS:" followed by a list of node names. Node arrays must be listed by elements, not the entire array. The nodes listed may not be AND or OR allocate nodes. The second line of the pair consists of "DEPARTURES:" followed by a list of expressions, one per node listed on the previous line.

In dialogue files only, prior to the specification of the run limits we may specify that an initial portion of the run is to be discarded, i.e., that only performance measures gathered after this initial portion will be discarded. The length of this portion is specified as a fraction, in percent, of the run limits (other than the CPU limit). The initial portion ends when the first of these fractions of the run length limits is reached. The run then ends when the first of the full limits is reached. The initial portion discarded is specified by a line of the form "INITIAL PORTION DISCARDED:" followed by a simulation independent expression. This expression should have a value in the interval [0,100).

The CPU limit is specified by a line of the form "LIMIT - CP SECONDS:" followed by a simulation independent expression. (Note that the keyword is "CP" so that "cpu" is available as a name.) This is only a rough limit because the simulation measures CPU time consumed after every 1000 events and pseudo-events (Appendix 7) and at other points considered significant. Thus more CPU time may be consumed than specified in this limit if the limit is reached between measurements.

The simulation dialogue following the initial state section for the example of Section 1.3 might be

```
INITIAL PORTION DISCARDED:10 /*percent*/
RUN LIMITS-
   SIMULATED TIME:3600
   EVENTS:50000
   QUEUES FOR DEPARTURE COUNTS:cssm1.memory
      DEPARTURES:              400
   QUEUES FOR DEPARTURE COUNTS:cssm1.iosys1.diskq cssm1.iosys2.diskq
      DEPARTURES:              2000              2000
   NODES FOR DEPARTURE COUNTS:cssm1.decrcycles
      DEPARTURES:             3000
LIMIT - CP SECONDS:5
```

12.2.2. Independent Replications

This section assumes that "replications" is specified on the confidence interval method line. With independent replications the simulation run is repeated several times (usually five to ten times) with each replication beginning in the same initial state. The only difference between the replications is that the random number streams are not reset at the beginning of the second and subsequent replications, so the replications are different due to statistical variability. (Section 12.3 discusses the random number streams of the RESQ simulation program.) The random number streams for the second replication begin where the streams for the first replication ended, the streams for the third replication begin where the streams for the second replication ended, etc.

After the confidence interval method line, the initial state of the network is specified, using the same syntax and semantics as a simulation without confidence intervals (Section 12.2.1).

After the initial state definition section, there are lines to specify the confidence level and the number of replications. The confidence level line consists of "CONFIDENCE LEVEL:" followed by a simulation independent expression for the confidence level in percent. A null reply is allowed for the confidence level prompt in interactive mode. The confidence level line is optional in dialogue files. If the confidence level is not specified, the default value of 90 (percent) is used. The number of replications line consists of "NUMBER OF REPLICATIONS:" followed by a simulation independent expression. The number of replications must be explicitly given.

The remainder of the simulation dialogue is essentially the same for replications as it is for simulation without confidence intervals. The "RUN LIMITS-" line is replaced by a "REPLIC LIMITS-" line. Otherwise the syntax is the same. The simulated time, event and departure limits are limits for each replication. A replication stops when the first of these limits is reached. The initial portion of each replication may be discarded, as with simulation without confidence intervals. The CPU limit is the limit for the total time spent on all replications. When the simulation stops, it may only be resumed if the CPU limit was reached

before the specified replications were completed. After the simulation stops, the replication limits may not be increased, nor may the number of replications be increased.

The simulation dialogue following the distribution specification for the example of Section 1.3, for the independent replications confidence interval method, might be

```
CONFIDENCE INTERVAL METHOD:replications
INITIAL STATE DEFINITION-
CHAIN:interactiv
   NODE LIST:terminals cssm1.setcmdtype
   INIT POP: users-2   2
CONFIDENCE LEVEL:95 /*percent*/
NUMBER OF REPLICATIONS:7
INITIAL PORTION DISCARDED:10 /*percent*/
REPLIC LIMITS-
   SIMULATED TIME:3600
   EVENTS:50000
   QUEUES FOR DEPARTURE COUNTS:cssm1.memory
      DEPARTURES:              400
   QUEUES FOR DEPARTURE COUNTS:cssm1.iosys1.diskq cssm1.iosys2.diskq
      DEPARTURES:              2000              2000
   NODES FOR DEPARTURE COUNTS:cssm1.decrcycles
      DEPARTURES:              3000
LIMIT - CP SECONDS:100
```

12.2.3. The Regenerative Method

This section assumes that "regenerative" is specified on the confidence interval method line. The regenerative method applies only to networks which *regenerate,* i.e., which return "frequently" (say, at least 10 times in a simulation run) to a state (usually the initial state) such that future behavior is independent of past behavior. With the example of Section 1.3, the initial state with all jobs at the terminals is a state with these characteristics for the parameters specified in Section 1.3. With other parameters, e.g., with very small "thinktime," that initial state might not occur sufficiently frequently. With a network consisting of only open chains, the state where the network is empty of jobs will often be a suitable choice of state. The state we have been discussing is called the "regeneration" state. It is usually the same as the initial state but may be a different state, as we discuss below.

With the regenerative method the simulation run is essentially the same as in simulation without confidence intervals, but the simulation program recognizes returns to the regeneration state. When the simulated system returns to the regeneration state, the program gathers information that will be used to estimate confidence intervals at the end of the simulation.

After the confidence interval method line, the regeneration and initial states of the network are specified, using syntax and semantics similar to simulation without confidence intervals (Section 12.2.1). The "INITIAL STATE DEFINITION-" line is replaced by "REGENERATION STATE DEFINITION-". Between the "NODE LIST:" and "INIT POP:" lines is inserted a "REGEN POP:" line. Except for the difference in keywords, this line has the same characteristics as the "INIT POP:" line. Since most node types consume zero simulated time and do not cause jobs to wait, non-zero numbers of job copies in the "REGEN POP:" line are only reasonable for classes, allocate nodes and fusion nodes. The simulation program only allows non-zero numbers of jobs in the "REGEN POP:" line for classes and "plain" allocate nodes. The values given by this line count both jobs and job copies, so for a closed chain the sum of the values in this line may be more than the chain

population. For example, for the model of Section 1.3 using the same initial state definition as in Section 12.2.1, the regeneration state section should be

```
REGENERATION STATE DEFINITION-
CHAIN:interactiv
   NODE LIST:terminals cssm1.setcmdtype cssm1.getmemory cssm1.cpu
   REGEN POP:users-2   0                2                2
   INIT POP: users-2   2                0                0
```

Here the initial state and the regeneration state are different, but the simulated system enters the regeneration state at simulated time zero (because the set node and allocate node take zero simulated time).

In general, the numbers of jobs and job copies at each node are not sufficient to rigorously define a regeneration state. Additional characteristics are defined by default in order to more rigorously define a regeneration state. Warning messages are issued when the state defined appears to the program to not be a rigorously defined regeneration state. Warnings are issued when

- A class has service time or work demand specified by an expression dependent on simulation variables or status functions or by a distribution not represented by exponential stages. (Exponential distributions, the BE distribution and the STANDARD distribution with coefficient of variation at least .5 are the only RESQ distributions represented by exponential stages. See Appendix 3.) Further, the regeneration state has a non-zero number of jobs at this class.

- A source has arrival time specified by an expression dependent on simulation variables or status functions or by a distribution not represented by exponential stages.

- Global variables are used.

- The regeneration state has a non-zero number of jobs at an allocate node. This warning only applies to queueing time distribution values other than mean queueing time. Regeneration states must be more rigorously defined for queueing time distributions.

When these messages are issued, the program proceeds with the simulation as if a regeneration state had been rigorously defined. The additional default characteristics of the regeneration state are

- Where service times and/or arrival times are represented by exponential stages, any times in progress are in the first stage in the regeneration state.

- At active queues where different orderings of the jobs in the queue are important (e.g., FCFS queueing discipline) the ordering of jobs of different classes is the same as at the first occurrence of the required numbers of jobs at all nodes.

- At passive queues the ordering of jobs of different allocate nodes and different numbers of tokens requested is the same as at the first occurrence of the required numbers of jobs at each node.

- CV(0) has the value one (1) for all open chains (see Section 9.1.2).

These warnings and default conditions are incomplete in the sense that there are states which will be accepted as rigorously defined regeneration states when in fact further conditions must be placed on the state definition to obtain a rigorously defined regeneration state.

After the regeneration state definition section, there is a line to specify the confidence level, as with independent replications. The confidence level line consists of "CONFIDENCE LEVEL:" followed by a simulation independent expression for the confidence level in percent. A null reply is allowed for the confidence level prompt in interactive mode. The confidence level line is optional in dialogue files. If the confidence level is not specified, the default value of 90 (percent) is used.

After the confidence level line is a required line to indicate whether the sequential stopping rule is to be used. The sequential stopping rule determines run length based on the confidence intervals determined at intermediate points in the run. The line consists of "SEQUENTIAL STOPPING RULE:" followed by "yes" or "no". We first consider the case without the sequential stopping rule, then the case with the sequential stopping rule.

If the "no" reply is given on the sequential stopping rule line, the remainder of the simulation dialogue is closely similar to the dialogue for simulation without confidence intervals. The "RUN LIMITS-" line is replaced by a "RUN GUIDELINES-" line. The periods between returns to the regeneration state are called "cycles." The values in the run guidelines are not firm limits because once one of these guidelines is reached, the simulation run will continue until either (1) the simulated system returns to the regeneration state, thus completing a regeneration cycle, or (2) the CPU limit is reached. The simulated time, event and departure lines are the same as with the run limits section for simulation without confidence intervals. After the "SIMULATED TIME:" line there may be a line specifying a limit for number of regeneration cycles for the run. This is truly a limit in that the simulated system will be returning to the regeneration state when the value is reached. The cycles line consists of "CYCLES:" followed by an expression for the number of cycles. The initial portion discarded line is not allowed with the regenerative method. The CPU limit line is the same as with simulation without confidence intervals. If the CPU limit is reached in the midst of a regeneration cycle, only the data from completed cycles will be used in the performance measure reports. When the simulation stops, it may be resumed as with simulation without confidence intervals. If this is done, and the simulation stopped because of the CPU limit, the simulation resumes in the midst of the incomplete regeneration cycle.

The simulation dialogue following the regeneration state definition, for the example of Section 1.3, might be

```
CONFIDENCE LEVEL:95 /*percent*/
SEQUENTIAL STOPPING RULE:no
RUN GUIDELINES-
   SIMULATED TIME:3600
   CYCLES:50
   EVENTS:50000
   QUEUES FOR DEPARTURE COUNTS:cssm1.memory
      DEPARTURES:              400
   QUEUES FOR DEPARTURE COUNTS:cssm1.iosys1.diskq cssm1.iosys2.diskq
      DEPARTURES:              2000                2000
   NODES FOR DEPARTURE COUNTS:cssm1.decrcycles
      DEPARTURES:              3000
LIMIT - CP SECONDS:100
```

If the sequential stopping rule is enabled, i.e., if the "yes" reply is given on the sequential stopping rule line, the simulation run will consist of one or more subruns, called "sampling periods." The user specifies the length of these sampling periods in a section corresponding to the run guidelines section. At the end of each sampling period, confidence intervals will be computed and evaluated with criteria specified by the user. If the criteria are satisfied, the simulation run stops. If the criteria are not satisfied, the simulation continues for at least one more sampling period. The criteria are basically prespecified widths for confidence intervals for certain queues and certain performance measures. In addition, the user may require that these width criteria be satisfied for several successive sampling periods.

After the sequential stopping rule line, there will be one or more triples of lines. The first line of a triple will be "QUEUES TO BE CHECKED:" followed by a list of names of queues. A queue name may be repeated in the list if width requirements are to be specified for more than one performance measure for that queue. The second line of a triple will be "MEASURES:" followed by a list of code, one code per queue name in the previous list. The allowed codes are

ut Utilization.

tp Throughput.

ql Mean queue length.

qld Queue length distribution.

qt Mean queueing time.

qtd Queueing time distribution.

tu Mean number of tokens in use (passive queues only).

tud Token use distribution (passive queues only).

tt Mean total number of tokens (passive queues only).

tud Total token distribution (passive queues only).

The distribution codes only apply if gathering of that distribution has previously been specified. Each gathered point of a listed distribution is checked and must satisfy the width criteria. The third line of the triple consists of "ALLOWED WIDTHS:" followed by a list of simulation independent expressions, one for each name on the first line of the triple. For the measures which can only have values in the [0,1] interval, utilization and the distribution measures, the width specified is absolute width in percent, i.e., the criterion is that $200 \times \delta$ be less than the specified width, where the confidence interval is $(p - \delta, p + \delta)$. For the other measures the width is relative width in percent, i.e., the criterion is that $200 \times \delta / p$ be less than the specified width. (Where p is zero, the criteria is not satisfied.)

After one or more triples have given the confidence interval width criteria, an additional requirement may be made that the width criteria be satisfied for several *successive* sampling periods. This requirement is specified by a line of the form "EXTRA SAMPLING PERIODS:" followed by a simulation independent expression. Specification of this requirement is optional; the default value is zero. The simulation will continue (assuming the CPU limit is not reached) until this number plus one successive sampling periods satisfy the width criteria.

The remainder of the dialogue is the same as with the regenerative method without the sequential stopping rule, except that the line "RUN GUIDELINES-" is replaced by "SAMPLING PERIOD GUIDELINES-". The sequential stopping rule should be used in a conservative manner, i.e., the sampling period guidelines should be specified with the intent that there be only a few, relatively long sampling periods, not many short sampling periods. A sampling period will continue until the first return to the regeneration state after one of these guidelines is reached, unless the CPU limit is reached first. If the the simulation stops because of the CPU limit, only data from completed regeneration cycles is used. When the simulation stops, it may be resumed, but only to increase the CPU limit or to increase the extra sampling period requirement. If this is done, and the simulation stopped because of the CPU limit, the simulation resumes in the midst of the incomplete regeneration cycle.

The simulation dialogue following the regeneration state definition, for the example of Section 1.3 with sequential stopping might be

```
CONFIDENCE LEVEL:95 /*percent*/
SEQUENTIAL STOPPING RULE:yes
   QUEUES TO BE CHECKED:cssm1.memory        cssm1.memory
      MEASURES:         qt                  qtd
      ALLOWED WIDTHS:   5 /*% - relative*/ 10 /*% - absolute*/
   QUEUES TO BE CHECKED:cssm1.cpuq          cssm1.iosys1.diskq
      MEASURES:         ut                  ut
      ALLOWED WIDTHS:   10 /*% - absolute*/ 10 /*% - absolute*/
   EXTRA SAMPLING PERIODS:1
SAMPLING PERIOD GUIDELINES-
   SIMULATED TIME:3600
   CYCLES:50
   EVENTS:50000
   QUEUES FOR DEPARTURE COUNTS:cssm1.memory
      DEPARTURES:              400
   QUEUES FOR DEPARTURE COUNTS:cssm1.iosys1.diskq cssm1.iosys2.diskq
      DEPARTURES:              2000               2000
   NODES FOR DEPARTURE COUNTS:cssm1.decrcycles
      DEPARTURES:              3000
LIMIT - CP SECONDS:100
```

12.2.4. The Spectral Method

This section assumes that "spectral" is specified on the confidence interval method line. Most methods in classical statistics for estimating confidence intervals depend on having items of data that are "independent and identically distributed." The method of independent replications achieves this "i.i.d." property by the protocol which repeats the simulation. The regenerative method depends on being able to observe the i.i.d. property during the simulation run. The spectral method does not depend on the i.i.d. property. Rather, it explicitly takes into consideration the correlation between data items in the simulation, e.g., the dependencies between successive queueing times for a given queue. This is done without user awareness, other than the availability of confidence intervals, so the dialogue for simulation using the the spectral method is essentially the same as simulation without confidence intervals. A sequential stopping rule is available with the spectral method, a slightly different rule than the one used with the regenerative method.

The spectral method requires substantial additional virtual storage *per performance measure, per queue or node,* for its confidence interval calculations, so confidence intervals are only available for mean queueing times and queueing time distributions, and then only for

queues and nodes specified by the user prior to the simulation. (The storage requirement for a given queue or node is on the order of 1600 bytes for mean queueing time, plus 1600 bytes for each point of the queueing time distribution.)

After the confidence interval method line, the initial state of the network is specified, using the same syntax and semantics as simulation without confidence intervals (Section 12.2.1). After the initial state definition section, there is an optional confidence level line, "CONFIDENCE LEVEL:" followed by a simulation independent expression giving confidence level in percent. As with the other confidence interval methods, the default is 90%. Then there is a line indicating whether or not the sequential stopping rule is to be used. As with the regenerative method, this line consists of "SEQUENTIAL STOPPING RULE:" followed by "yes" or "no." We first consider the case without the sequential stopping rule, then the case with the sequential stopping rule.

If the "no" reply is given on the sequential stopping rule line, the next part of the simulation dialogue consists of (optional) pairs of lines for listing queues which are to have confidence intervals computed. The first line of a pair consists of "CONFIDENCE INTERVAL QUEUES:" followed by a list of names of queues. Names may be repeated if both mean queueing time and queueing time distribution confidence intervals are to be computed for the same queue. The second line of a pair consists of "MEASURES:" followed by a list of codes, either "qt" for mean queueing time or "qtd" for queueing time distribution, one code per name in the previous line. The qtd code only applies if gathering of that distribution was previously specified. After the pairs of lines for queues follow (optional) pairs of lines for nodes. The first line of a pair consists of "CONFIDENCE INTERVALQUEUES:" followed by a list of names of nodes. Only names of classes and "plain" allocate nodes may be listed. The second line of a pair is as with the queue pairs. The remainder of the dialogue is as with simulation without confidence intervals, beginning with the optional initial portion discarded line.

The simulation dialogue following the initial state definition, for the example of Section 1.3, might be

```
CONFIDENCE LEVEL:95 /*percent*/
SEQUENTIAL STOPPING RULE:no
CONFIDENCE INTERVAL QUEUES:cssm1.memory cssm1.memory
      MEASURES:             qt            qtd
CONFIDENCE INTERVAL NODES:cssm1.cpu
      MEASURES:           qt
INITIAL PORTION DISCARDED:10 /*percent*/
RUN LIMITS-
   SIMULATED TIME:3600
   EVENTS:50000
   QUEUES FOR DEPARTURE COUNTS:cssm1.memory
      DEPARTURES:              400
   QUEUES FOR DEPARTURE COUNTS:cssm1.iosys1.diskq cssm1.iosys2.diskq
      DEPARTURES:              2000                2000
   NODES FOR DEPARTURE COUNTS:cssm1.decrcycles
      DEPARTURES:              3000
LIMIT - CP SECONDS:100
```

If the sequential stopping rule is enabled, i.e., if the "yes" reply is given on the sequential stopping rule line, the simulation run will consist of one or more subruns, called "sampling periods." Unlike the regenerative method, where these periods are of approximately equal length, with the spectral method these periods are such that the total run length increases by

roughly 50% with each sampling period. As with the regenerative method, after each sampling period user specified criteria are used to determine whether to stop the run. If the criteria are not satisfied, the simulation continues for at least one more sampling period. The criteria are basically prespecified widths for confidence intervals for certain performance measures and certain queues and nodes. In addition, the user may require that these width criteria be satisfied for several successive sampling periods.

After the sequential stopping rule line, there will be two groups of triples of lines corresponding to the two groups of pairs of lines for the queues/nodes for confidence intervals as in the spectral method without the sequential stopping rule. The first two lines of each triple are the same as the pairs of lines. The third line of the triple consists of "ALLOWED WIDTHS:" followed by a list of simulation independent expressions, one for each name on the first line of the triple. For the queueing time distribution, the width specified is absolute width in percent, i.e., the criterion is that $200\times\delta$ be less than the specified width, where the confidence interval is $(p - \delta, p + \delta)$. For mean queueing time the width is relative width in percent, i.e., the criterion is that $200\times\delta/p$ be less than the specified width. (Where p is zero, the criteria is not satisfied.)

After these triples have given the confidence interval width criteria, an additional requirement may be made that the width criteria be satisfied for several *successive* sampling periods. This requirement is specified by a line of the form "EXTRA SAMPLING PERIODS:" followed by a simulation independent expression. Specification of this requirement is optional; the default value is zero. The simulation will continue (assuming the CPU limit is not reached) until this number plus one successive sampling periods satisfy the width criteria.

The remainder of the dialogue is the same as with the spectral method without the sequential stopping rule, except that the line "RUN LIMITS-" is replaced by "INITIAL PERIOD LIMITS-". The limits specified are for the initial sampling period. These limits are increased by 50% at the beginning of each sampling period and are then used as limits for the total length of the run, not the length of the sampling period. The sequential stopping rule should be used in a conservative manner, i.e., the initial period limits should be specified with the intent that there be only a few, relatively long sampling periods, not many short sampling periods. If it is specified that an initial portion of the run is to be discarded, only this portion of the initial sampling period is discarded. When the simulation stops, it may be resumed, but only to increase the CPU limit or to increase the extra sampling period requirement.

The simulation dialogue following the initial state definition, for the example of Section 1.3 with sequential stopping might be

```
CONFIDENCE LEVEL:95 /*percent*/
SEQUENTIAL STOPPING RULE:yes
   CONFIDENCE INTERVAL QUEUES:cssm1.memory        cssm1.memory
      MEASURES:               qt                  qtd
      ALLOWED WIDTHS:         5 /*% - relative*/ 10 /*% - absolute*/
   CONFIDENCE INTERVAL NODES:cssm1.cpu
      MEASURES:              qt
      ALLOWED WIDTHS:        10 /*% - absolute*/
    EXTRA SAMPLING PERIODS:1
INITIAL PORTION DISCARDED:10 /*percent of initial sampling period*/
INITIAL PERIOD LIMITS-
   SIMULATED TIME:3600
   EVENTS:50000
   QUEUES FOR DEPARTURE COUNTS:cssm1.memory
      DEPARTURES:              400
```

```
QUEUES FOR DEPARTURE COUNTS:cssm1.iosys1.diskq cssm1.iosys2.diskq
   DEPARTURES:              2000                2000
NODES FOR DEPARTURE COUNTS:cssm1.decrcycles
   DEPARTURES:              3000
LIMIT - CP SECONDS:100
```

12.3. Random Number Generation

The topics of this section affect only a single line of dialogue. This line appears only in dialogue files and is optional in dialogue files. Before discussing this line we discuss the generation of (pseudo) random numbers in the simulation program.

Set	Sources	Active	Routing	Passive	Set Nodes
1	377003613	1267310126	1976418161	150006407	288727775
2	648473574	1741371275	35067978	1633650593	1499601820
3	1396717869	886499692	400884188	751601611	2136214308
4	2027350275	1014119573	1895732964	1410990605	1197972807
5	1356162430	933913228	1904749580	1262214427	1888007825
6	1752629996	2082204497	1301700180	645360044	686553263
7	645806097	920168983	63685808	1504645702	747119178
8	201331468	1079618777	936615625	1063375004	154337000
9	1393552473	1888797415	110322717	941885586	136758808
10	1966641861	1002901030	1029730003	1753135176	9182540
11	711072531	1582733583	251900732	253642018	303111010
12	769795447	254293472	725094089	1701685042	154232008
13	1074543187	1095895189	828842333	1448665492	921093990
14	1933483444	219529399	1471230052	1034856864	1684263351
15	625102656	1706847402	1703522097	428280431	1166344707
16	1116874679	1951007719	1356420548	259758456	1167753617
17	1442211901	1169002398	1670372925	600732272	1374693082
18	989455196	1482199345	437765009	704726097	1812641667
19	1996695068	1976077334	39279049	398944698	502455872
20	1850124212	775245191	2123613511	114386769	857532898

Table 12.1 - Seeds for Random Number Streams

There are five random number streams in the simulation. Separate streams are used for sources, for active queues, for routing decisions, for passive queues and for set nodes. There are twenty sets of five seeds for initializing these streams. In a dialogue file, a line may be inserted after the CPU limit line to indicate which set of five seeds is to be used. Other than choosing a set of seeds, the user has no control over random number streams. The line consists of "SEED:" followed by a simulation independent expression. This expression should be an integer between 1 and 20. If the line is omitted or the expression has an inappropriate value, set 1 is used. Table 12.1 gives the 20 sets of five seeds. Random integers in the simulation are generated by a function of the form

$$X_n = 7^5 X_{n-1} \text{ modulo } 2^{31}-1$$

where X_n is the desired random integer and X_{n-1} is the previous random integer of the stream or the seed of the stream if no previous random numbers have been obtained from the stream. ($7^5 = 16807$ and $2^{31}-1 = 2147483647$.) The values in Table 12.1 were obtained from this generator by taking every hundred thousandth random integer starting at 377003613. Reading

horizontally, the table entries are two million values apart. Uniform random numbers on the interval (0,1) are obtained by dividing the random integer obtained from the generator by $2^{31}-1$. Exponential random numbers are obtained by taking the natural logarithm of a uniform random number on the interval (0,1). The logarithm is negated and then multiplied by the desired mean of the exponential distribution. All random numbers in the simulation are obtained from simple functions of uniform and exponential random numbers.

12.4. Simulation Trace

Simulation trace lets the user know what happens during (a portion of) a run. This is useful to the user in developing (debugging) a model. If the user suspects an error in the simulation program itself, trace can be used to either confirm or deny this suspicion.

Trace specification is the last section of the model definition. The first line of trace specification consists of "TRACE:" followed by "yes" or "no". If no trace is indicated (by "no"), the model definition is completed by a line containing only "END".

If interactive mode is used and trace is indicated (by "yes"), two additional prompts will be given. The two additional prompts are "JOB MOVEMENT:" and "QUEUES:". The reply to JOB MOVEMENT: must be either "yes" or "no." The reply to QUEUES: may be "yes" or "no" or a list of queue names. If the reply is "yes" then all queues will have "queue trace." In interactive mode the model definition is complete after the "QUEUES:" line.

In dialogue files other forms of trace may be specified and trace may be selectively enabled and disabled for portions of a run. After the "TRACE:yes" line comes a line to indicate whether trace is initially on or off. This line consists of "INITIALLY ON:" followed by "yes" or "no". Then there are two optional sections for specifying when trace will be turned on during the run and when trace will be turned off during the run. If independent replications are used for confidence intervals, these sections apply to each replication. The syntax and capabilities for turning trace on and off parallel the dialogue sections for specifying limits or guidelines described in Section 12.2. The section for turning trace on begins with a line "TURN TRACE ON-". Following that are (optional) lines for simulated time, regeneration cycles (if the regenerative method is used for confidence intervals), simulated events, queue departure counts and node departure counts. The section for turning trace off is the same syntactically except that the first line is "TURN TRACE OFF-". After these sections are two lines for job movement trace and queue trace, corresponding to the interactive prompts described above. Then there are three additional lines: "EVENT HANDLING:" followed by "yes" or "no", "EVENT LIST:" followed by "yes" or "no" and "SNAPSHOTS:" followed by "yes" or "no". An "END" line completes the model definition.

For example, we might have the following:

```
TRACE:yes
   INITIALLY ON:yes
   TURN TRACE ON -
      SIMULATED TIME:3.5
      CYCLES:
      EVENTS:
      QUEUES FOR DEPARTURE COUNTS:cssm1.memory cssm1.cpuq
         DEPARTURES:              500          1300
      QUEUES FOR DEPARTURE COUNTS:
      NODES FOR DEPARTURE COUNTS:
   TURN TRACE OFF -
```

```
        SIMULATED TIME:4.5
        CYCLES:
        EVENTS:
        QUEUES FOR DEPARTURE COUNTS:cssm1.memory cssm1.cpuq
           DEPARTURES:                510          1305
        QUEUES FOR DEPARTURE COUNTS:
        NODES FOR DEPARTURE COUNTS:
     JOB MOVEMENT:yes
     QUEUES:memoryq cpuq
     EVENT HANDLING:yes
     EVENT LIST:no
     SNAPSHOTS:no
END /*of model*/
```

When the simulated time option is used, it will only have an effect if an event occurs at exactly the specified time. When several options are used to turn the trace on, they will each be enforced if possible, i.e., the trace will be turned on (if it is not already on) at the occurrence of each specified condition. Similarly, several options to turn trace off will each be enforced if possible. Only one departure count to turn trace on may be specified for a given queue or node, and only one departure count may be specified to turn trace off for a given queue or node.

The special global variables discussed in Appendix 2 may also be used to control trace. In addition to the trace capabilities described in this section, Appendix 3 describes the PRINT function which may be used for observing values of numeric expressions.

We now give examples of the job movement trace, the queue trace and the event trace. *Event list and snapshot trace produce large amounts of output.* It is usually inappropriate to use these forms of trace. We will briefly discuss these two forms of trace at the end of this section.

The job movement trace, which shows the movement of jobs through the network, is usually the most important. We illustrate some of the job movement trace for arbitrary portions of models which illustrate by example most of the trace output for job movement. The trace output we show is that from the RQ2PRNT file, but the same output is also displayed at the terminal during simulation. We will intersperse explanations before pieces of trace output.

Sections of trace output are labeled at the beginning by the name of the procedure that produced the output. The procedure that handles the simulation events and timing is called "SMULAT."

```
RESQ2 VERSION DATE: JANUARY 29, 1982 -  TIME: 17:45:03  DATE: 01/29/82
MODEL:LOOP

SMULAT -- SIMULATION BEGINS...
```

The procedure that handles routing is called "ARRIVE." Jobs in the network are numbered in the order of their creation. The following says that job 1 is the first departure from a source named "S." It then gives the current time and number of events. Then it gives the current network population, both in terms of true jobs and copies of jobs holding tokens at allocate nodes. Then the destinations are considered in order until one is selected according to its routing probability or predicate. In this case the first destination is selected.

```
ARRIVE -- JOB              1 DEPARTURE       1 FROM S(SOURCE)
          CURRENT TIME:  5.6301821E-02 NUMBER OF EVENTS:          1
          POPULATION:     1 JOBS,    0 JOB COPIES
          DESTINATION, CONDITION:
          BEGINRT(ALLOCATE), 0.178339 < 1.000000
```

Jobs are not considered to be "departures" from allocate nodes until they release or destroy their tokens (except for AND and OR allocate nodes) so even though job 1 leaves "beginrt," it is not counted as a departure. The trace shows no departure counts for AND and OR allocate nodes. Since job 1 now holds tokens, a list of nodes where tokens are held is now provided by ARRIVE.

```
ARRIVE -- JOB              1 DEPARTURE       0 FROM BEGINRT(ALLOCATE)
          CURRENT TIME:  5.6301821E-02 NUMBER OF EVENTS:          1
          POPULATION:     1 JOBS,    1 JOB COPIES
          TOKENS HELD:           1 AT BEGINRT
          DESTINATION, CONDITION:
          SET_MSG_L(SET), 0.343924 < 1.000000
```

Set nodes must evaluate expressions which are dependent on values not known until simulation time, e.g., global variables and results of status functions. Expressions are stored internally in prefix ("reverse Polish") notation. Procedure EXPRT serves only to print the prefix form of the expression. The assignment jv(pkt__lng)=standard(totlength,1) is to be evaluated.

```
EXPRT --  = SUB1 JV PKT_LENG   STANDARD ; , TOTLENGTH   1
```

If a chain variable or global variable is assigned a value at a set node, procedure SETNOD will print this value. However, ARRIVE prints the values of all non-zero job variables, so SETNOD does not print values of job variables it changes. All of the routing so far has not involved decisions, i.e., there was only one possible destination. In general routing may involve probabilities mixed with predicates, as discussed in Section 9.1.4.

```
ARRIVE -- JOB              1 DEPARTURE       1 FROM SET_MSG_L(SET)
          CURRENT TIME:  5.6301821E-02 NUMBER OF EVENTS:          1
          POPULATION:     1 JOBS,    1 JOB COPIES
          JV'S¬=0:     1:  2.670E+02
          TOKENS HELD:           1 AT BEGINRT
          DESTINATION, CONDITION:
          DEST1(SET), 0.334520 < 0.250000
          DEST2(SET), 0.084520 < 0.250000
```

The following is for jv(msg__dest)=discrete(1,1/3;3,1/3;4,1/3). The internal conventions for commas and semi-colons are not the same as the external conventions. Internally, a semicolon is always represented by a "; ," pair which precedes a list element, e.g., "1,1/3". The symbol "EOX" is used to indicate the end of a list separated by semicolons.

```
EXPRT --  = SUB1 JV MSG_DEST   DISCRETE ; ,  1 /  1  3 ; ,  3 /  1  3 ;
          ,  4 /  1  3 EOX
```

EXPRT is also used when predicates are evaluated, e.g., to evaluate the predicate if(jv(pkt__leng)<=240).

```
ARRIVE -- JOB              1 DEPARTURE       1 FROM DEST2(SET)
          CURRENT TIME:  5.6301821E-02 NUMBER OF EVENTS:          1
```

```
          POPULATION:     1 JOBS,    1 JOB COPIES
          JV'S¬=0:     0:  4.000E+00    1:  2.670E+02
          TOKENS HELD:           1 AT BEGINRT
          DESTINATION, CONDITION:
          C2(CLASS), (PREDICATE)
EXPRT --  <= SUB1 JV PKT_LENG     240
          SEPARATE2(FISSION), 0.278551 < 1.000000
```

Routine FISSN handles fission nodes. It gives the identities of children it creates.

```
FISSN --  PARENT IS         1, CHILD IS          2
```

When a job has relatives, ARRIVE will list immediate relatives (but not grandparents, grandchildren, etc.).

```
ARRIVE -- JOB              1 DEPARTURE       1 FROM SEPARATE2(FISSION)
          CURRENT TIME:  5.6301821E-02 NUMBER OF EVENTS:          1
          POPULATION:     2 JOBS,    1 JOB COPIES
          JV'S¬=0:     0:  4.000E+00    1:  2.670E+02
          TOKENS HELD:           1 AT BEGINRT
          RELATIVES:  CHILD          2 AT SEPARATE2
          DESTINATION, CONDITION:
          DEC_MSG_L2(SET), (PREDICATE)
EXPRT --  = SUB1 JV PKT_LENG   - SUB1 JV PKT_LENG     240

ARRIVE -- JOB              1 DEPARTURE       1 FROM DEC_MSG_L2(SET)
          CURRENT TIME:  5.6301821E-02 NUMBER OF EVENTS:          1
          POPULATION:     2 JOBS,    1 JOB COPIES
          JV'S¬=0:     0:  4.000E+00    1:  2.698E+01
          TOKENS HELD:           1 AT BEGINRT
          RELATIVES:  CHILD          2 AT SEPARATE2
          DESTINATION, CONDITION:
          C2(CLASS), (PREDICATE)
EXPRT --  <= SUB1 JV PKT_LENG     240
```

The service time expression at "C2" is printed by EXPRT because it involves values which cannot be determined before simulation.

```
EXPRT --  STANDARD ; , / SUB1 JV PKT_LENG    CAPACITY     0
```

Note that job 1 keeps moving until it reaches a class, as specified in the rules in Appendix 7, and that job 2 moves at the same clock time after job 1 stops. Also: children are not considered as departures from fission nodes; children get the same job variables as their parents.

```
ARRIVE -- JOB              2 DEPARTURE       1 FROM SEPARATE2(FISSION)
          CURRENT TIME:  5.6301821E-02 NUMBER OF EVENTS:          1
          POPULATION:     2 JOBS,    1 JOB COPIES
          JV'S¬=0:     0:  4.000E+00    1:  2.670E+02
          RELATIVES:  PARENT         1 AT C2
          DESTINATION, CONDITION:
          SET_PKT_L2(SET), (PREDICATE)
EXPRT --  = SUB1 JV PKT_LENG     240
  ...
```

When a job arrives at a fusion node and finds no relatives, there is no special indication.

```
ARRIVE -- JOB              1 DEPARTURE       1 FROM C3(CLASS)
          CURRENT TIME:  6.7541439E-02 NUMBER OF EVENTS:          3
          POPULATION:     2 JOBS,    1 JOB COPIES
          JV'S¬=0:     0:  4.000E+00    1:  2.698E+01
          TOKENS HELD:            1 AT BEGINRT
          RELATIVES:  CHILD          2 AT C2
          DESTINATION, CONDITION:
          ASSEMBLE(FUSION), (PREDICATE)
EXPRT --  = SUB1 JV MSG_DEST     4
```

But when a job arrives at a fusion node where a relative is waiting, one will be destroyed. Routine FUSN uses routine SNKFUS to destroy the job.

```
   ...

ARRIVE -- JOB              2 DEPARTURE       2 FROM C3(CLASS)
          CURRENT TIME:  1.6192162E-01 NUMBER OF EVENTS:          6
          POPULATION:     5 JOBS,    2 JOB COPIES
          JV'S¬=0:     0:  4.000E+00    1:  2.400E+02
          RELATIVES:  PARENT          1 AT ASSEMBLE
          DESTINATION, CONDITION:
          ASSEMBLE(FUSION), (PREDICATE)
EXPRT --  = SUB1 JV MSG_DEST     4
FUSN --   LOOKING FOR RELATIVES OF JOB          2
          FOUND PARENT OF JOB          2          PARENT=          1
SNKFUS -- JOB          2 AT NODE ASSEMBLE
```

Routine SNKFUS also handles sinks.

```
ARRIVE -- JOB              1 DEPARTURE       1 FROM ASSEMBLE(FUSION)
          CURRENT TIME:  1.6192162E-01 NUMBER OF EVENTS:          6
          POPULATION:     4 JOBS,    2 JOB COPIES
          JV'S¬=0:     0:  4.000E+00    1:  2.698E+01
          TOKENS HELD:            1 AT BEGINRT
          DESTINATION, CONDITION:
          SINK(SINK), 0.638042 < 1.000000
SNKFUS -- JOB          1 AT NODE SINK
  ...
```

The following is the form for initial placement of jobs, at the beginning of a run or replication. A job is initially placed at node "C2POLL." Since there is more than one chain, ARRIVE gives the number of jobs in each chain.

```
ARRIVE -- JOB              1
          CURRENT TIME:  0.0000000E+00 NUMBER OF EVENTS:          0
          POPULATION:     1 JOBS,    0 JOB COPIES(    0 JOBS IN CHMSG    1 J
          OBS IN CHPOLL)
          DESTINATION, CONDITION:
          C2POLL(CLASS), 0.178339 < 1.000000

ARRIVE -- JOB              1 DEPARTURE       1 FROM C2POLL(CLASS)
          CURRENT TIME:  3.9999998E-01 NUMBER OF EVENTS:          1
```

```
POPULATION:     1 JOBS,    0 JOB COPIES(    0 JOBS IN CHMSG    1 J
OBS IN CHPOLL)
DESTINATION, CONDITION:
POLL1(CREATE), 0.343924 < 1.000000
```

...

The following illustrates output for queue trace only. If job movement trace were also enabled, the two would be interleaved. Procedure ALLCTE handles allocate nodes.

```
ALLCTE -- JOB          1 AT NODE BEGINRT QUEUE RTQ TOKEN REQUEST          1
```

When ALLCTE is through, it calls PQTRAC to list the entire queue, in order. The value −1 is widely used in RESQ2 to represent "undefined." Since "RTQ" is not a priority queue, each job has undefined priority. The column "TOKNS" lists the number of tokens requested. The column "HELD?" indicates whether or not the job holds these tokens by 1 or 0, respectively. (With priority passive queues, allocation of tokens is handled by SMULAT, which will call PQTRAC after it tries to allocate tokens to a queue. See Appendix 7.)

```
PQTRAC --     JOB  NODE                      PRTY TOKNS HELD?      Q_PTR
                1  BEGINRT                     -1     1     1
PQTRAC -- RTQ TOKENS:2147479808 TOKENS AVAILABLE:2147479807
```

Procedure SERARR handles arrivals at active queues. The "service request" will usually be the service time, unless (1) the service time is sampled by stages for the regenerative method (Appendix 7) or (2) variable rate or heterogeneous servers are involved, in which cases servers are treated explicitly in the trace output.

```
SERARR -- JOB          1 AT CLASS C2 QUEUE Q2 SERVICE REQUEST  5.620E-03
```

When SERARR is through, it calls AQTRAC to list the whole queue. The time given by AQTRAC is the remaining service time. "DSTG" is only meaningful when the distribution is sampled by stages for the regenerative method, in which case it is the current distribution stage.

```
AQTRAC --     TIME       JOB  NODE                    PRTY  DSTG      Q_PTR
        5.620E-03          1  C2                        -1     0
AQTRAC -- Q2 SERVERS:          1 SERVERS AVAILABLE:          0
SERARR -- JOB          2 AT CLASS C2 QUEUE Q2 SERVICE REQUEST  5.000E-02
AQTRAC --     TIME       JOB  NODE                    PRTY  DSTG      Q_PTR
        5.620E-03          1  C2                        -1     0          2
        5.000E-02          2  C2                        -1     0
AQTRAC -- Q2 SERVERS:          1 SERVERS AVAILABLE:          0
```

Routine COMPLT handles completion of service times. It also calls AQTRAC when it is done.

```
COMPLT -- JOB          1 AT CLASS C2 QUEUE Q2
AQTRAC --     TIME       JOB  NODE                    PRTY  DSTG      Q_PTR
        5.000E-02          2  C2                        -1     0
AQTRAC -- Q2 SERVERS:          1 SERVERS AVAILABLE:          0
```

...

When SNKFUS, acting for a sink or fusion node, must release tokens, it prints a message. When it is done with a particular queue (it may have to release tokens at several queues), it calls PQTRAC. RELEAS (release nodes), DSTROY (destroy nodes) and CREATE (create nodes) behave similarly.

```
  ...
SNKFUS -- COPY          1 AT NODE BEGINRT QUEUE RTQ
PQTRAC --    JOB  NODE                      PRTY TOKNS HELD?      Q_PTR
               3  BEGINRT                     -1     1      1
PQTRAC -- RTQ TOKENS:2147479808 TOKENS AVAILABLE:2147479807
  ...
```

The event handling trace is oriented toward the internal mechanics of the simulation run. The following examples show interleaved job movement and event trace. If other kinds of trace were enabled, they would be interleaved with this trace. One feature of event handling trace is that routines SMULAT and ARRIVE will print current CPU time when they check it.

```
SMULAT -- ACCUMULATED CP SECONDS =  0.000E+00
```

The routine CHECK is used to determine whether the system is in the regeneration state (assuming the regenerative method is used). CHECK has three major sections, which determine, in order, whether the open chains have the proper populations, whether any sources with the BE distribution are in their first stage and whether the nodes have the proper numbers of jobs. (Additional conditions are also checked but not explicitly reported.) CHECK reports its successful findings and its overall determination, 1 or 0 depending on whether or not, respectively, the system is in the regeneration state.

```
CHECK -- CYCLE END? CHAIN POPS ACCEPTED. SOURCE STAGES ACCEPTED.
         NODE POPS ACCEPTED. RESULT=1
```

Procedure EOSRST (End Of SubRun STate) is used in a variety of situations which delineate major portions of a simulation. With the regenerative method, EOSRST is called every time the system is in the regeneration state. EOSRST takes note of the beginning and ends of sampling periods for the sequential stopping rule. (When the stopping rule is not enabled, EOSRST considers the whole run to be a sampling period.)

```
EOSRST -- CYCLES          0 LIMIT 2147483647 BEGINNING SAMPLING PERIOD
```

SMULAT reports each event it handles. Event handling is discussed in Appendix 7. First we have a source arrival.

```
SMULAT -- NO. EVENTS          1 TIME   5.6301821E-02(SOURCE) SOURCE S

ARRIVE -- JOB             1 DEPARTURE      1 FROM S(SOURCE)
          CURRENT TIME:  5.6301821E-02 NUMBER OF EVENTS:          1
          POPULATION:    1 JOBS,   0 JOB COPIES
          DESTINATION, CONDITION:
          BEGINRT(ALLOCATE), 0.178339 < 1.000000

ARRIVE -- JOB             1 DEPARTURE      0 FROM BEGINRT(ALLOCATE)
          CURRENT TIME:  5.6301821E-02 NUMBER OF EVENTS:          1
          POPULATION:    1 JOBS,   1 JOB COPIES
          TOKENS HELD:          1 AT BEGINRT
          DESTINATION, CONDITION:
```

```
          SET_MSG_L(SET), 0.343924 < 1.000000
EXPRT --  = SUB1 JV PKT_LENG   STANDARD ; , TOTLENGTH   1

ARRIVE -- JOB             1 DEPARTURE      1 FROM SET_MSG_L(SET)
          CURRENT TIME:  5.6301821E-02 NUMBER OF EVENTS:          1
          POPULATION:    1 JOBS,   1 JOB COPIES
          JV'S¬=0:    1:  2.670E+02
          TOKENS HELD:          1 AT BEGINRT
          DESTINATION, CONDITION:
          DEST1(SET), 0.334520 < 0.250000
          DEST2(SET), 0.084520 < 0.250000
EXPRT --  = SUB1 JV MSG_DEST   DISCRETE ; ,  1 /  1  3 ; ,  3 /  1  3 ;
          ,  4 /  1  3 EOX

ARRIVE -- JOB             1 DEPARTURE      1 FROM DEST2(SET)
          CURRENT TIME:  5.6301821E-02 NUMBER OF EVENTS:          1
          POPULATION:    1 JOBS,   1 JOB COPIES
          JV'S¬=0:    0:  4.000E+00   1:  2.670E+02
          TOKENS HELD:          1 AT BEGINRT
          DESTINATION, CONDITION:
          C2(CLASS), (PREDICATE)
EXPRT --  <= SUB1 JV PKT_LENG    240
          SEPARATE2(FISSION), 0.278551 < 1.000000
FISSN -- PARENT IS          1, CHILD IS          2

ARRIVE -- JOB             1 DEPARTURE      1 FROM SEPARATE2(FISSION)
          CURRENT TIME:  5.6301821E-02 NUMBER OF EVENTS:          1
          POPULATION:    2 JOBS,   1 JOB COPIES
          JV'S¬=0:    0:  4.000E+00   1:  2.670E+02
          TOKENS HELD:          1 AT BEGINRT
          RELATIVES:  CHILD         2 AT SEPARATE2
          DESTINATION, CONDITION:
          DEC_MSG_L2(SET), (PREDICATE)
EXPRT --  = SUB1 JV PKT_LENG   - SUB1 JV PKT_LENG    240

ARRIVE -- JOB             1 DEPARTURE      1 FROM DEC_MSG_L2(SET)
          CURRENT TIME:  5.6301821E-02 NUMBER OF EVENTS:          1
          POPULATION:    2 JOBS,   1 JOB COPIES
          JV'S¬=0:    0:  4.000E+00   1:  2.698E+01
          TOKENS HELD:          1 AT BEGINRT
          RELATIVES:  CHILD         2 AT SEPARATE2
          DESTINATION, CONDITION:
          C2(CLASS), (PREDICATE)
EXPRT --  <= SUB1 JV PKT_LENG    240
EXPRT --  STANDARD ; , / SUB1 JV PKT_LENG   CAPACITY    0
```

After the job that arrived from the source stops moving, a child it created at a fission node starts moving with a pseudo-arrival event.

```
SMULAT -- NO. EVENTS          1 TIME  5.6301821E-02(PSEUDO) JOB          2

ARRIVE -- JOB             2 DEPARTURE      1 FROM SEPARATE2(FISSION)
          CURRENT TIME:  5.6301821E-02 NUMBER OF EVENTS:          1
          POPULATION:    2 JOBS,   1 JOB COPIES
```

```
          JV'S¬=0:     0:  4.000E+00    1:  2.670E+02
          RELATIVES:  PARENT          1 AT C2
          DESTINATION, CONDITION:
          SET_PKT_L2(SET), (PREDICATE)
EXPRT --  = SUB1 JV PKT_LENG     240

ARRIVE -- JOB              2 DEPARTURE       1 FROM SET_PKT_L2(SET)
          CURRENT TIME:  5.6301821E-02 NUMBER OF EVENTS:          1
          POPULATION:     2 JOBS,    1 JOB COPIES
          JV'S¬=0:     0:  4.000E+00    1:  2.400E+02
          RELATIVES:  PARENT          1 AT C2
          DESTINATION, CONDITION:
          C2(CLASS), 0.280669 < 1.000000
EXPRT --  STANDARD ; , / SUB1 JV PKT_LENG    CAPACITY     0
```

CHECK is called before real events are handled, but not before pseudo-arrivals or passive queue allocation attempts.

```
CHECK -- CYCLE END? RESULT=0
```

The next event is a service completion.

```
SMULAT -- NO. EVENTS          2 TIME  6.1921630E-02(SERVER) JOB          1
          QUEUE Q2

ARRIVE -- JOB              1 DEPARTURE       1 FROM C2(CLASS)
          CURRENT TIME:  6.1921630E-02 NUMBER OF EVENTS:          2
          POPULATION:     2 JOBS,    1 JOB COPIES
          JV'S¬=0:     0:  4.000E+00    1:  2.698E+01
          TOKENS HELD:           1 AT BEGINRT
          RELATIVES:  CHILD           2 AT C2
          DESTINATION, CONDITION:
          ASSEMBLE(FUSION), (PREDICATE)
EXPRT --  = SUB1 JV MSG_DEST     3
          C3(CLASS), 0.197507 < 1.000000
EXPRT --  STANDARD ; , / SUB1 JV PKT_LENG    CAPACITY     0
  ...
```

The following shows the form for the passive queue allocation event.

```
SMULAT -- NO. EVENTS          1 TIME  3.9999998E-01(PRTYPQ) QUEUE POLL1Q
```

With replications, the routine APLOMB indicates the beginning of each replication.

```
APLOMB -- BEGIN REPLICATION          1
```

EOSRST will indicate the end of the initial portion of a run or replication if that portion has been specified to be discarded.

```
EOSRST -- END DISCARDED PORTION
```

EOSRST will indicate the end of a replication, including the limit(s) which caused it to end.

```
EOSRST -- END REPLICATION          1
EOSRST -- REPLICATION   1: RTQ DEPARTURE LIMIT
```

If event list trace is enabled, then the entire event list is shown by procedures ADEVNT or REMVEV every time an event is added or removed by a procedure other than SMULAT. (SMULAT only removes events from the list to handle them. Other procedures must remove events to handle preemption, to handle changes in queue length with PS, to handle changes in service rate and to handle changes in source rates.) If snapshot trace is enabled, then before any kind of event (including pseudo-arrival and passive queue events) is handled, the routine SNPSHT lists the numbers of jobs at each node and queue.

13. THE EVAL AND EVALT COMMANDS

This section covers basic usage of the EVAL command for model solution, and EVALT, a substitute for the EVAL command for use with the USER numeric function (Appendix 3). PL/I embedding (Section 14) may be used for model solution instead of either of these commands. Appendix 6 covers the error messages produced by the EVAL and EVALT commands.

13.1. EVAL Command

Before issuing the EVAL command, the user should be sure that his or her virtual machine has sufficient storage, that the virtual machine has access to the mini-disks containing the RESQ system files and the PL/I run time library, and that sufficient loader table space is provided. These steps typically will be the same as with the SETUP command (see Section 2.1), except that the EVAL command usually requires more virtual storage, and need to be taken only the first time RESQ is used, provided appropriate modifications are made to the CP directory and/or PROFILE EXEC.

Depending on the particular model and the sizes specified (perhaps by default) for internal dynamic storage areas, the EVAL command will typically require roughly 1300K of virtual storage. With some models and sizes for storage areas, 1100K or less may be sufficient, while for other situations 1300K will be insufficient. Additional information in this regard is given in Section 13.3.

The EVAL command may be issued without an argument, as in the example in Section 1. When issued without an argument, EVAL will prompt for a model name. Alternatively, EVAL may be issued with one or more arguments, the first of which is interpreted as the model name. Once the model name is established, the EVAL command is the same whether the model name was obtained from a prompt or an argument.

The EVAL command is oriented toward an interactive prompting mode. This is often the most effective mode because of the capabilities for selective examination of performance measures, for run continuation and for repeated execution with different parameters values. However, it is possible to provide replies prior to anticipated prompts, either from a file or as arguments to the EVAL command. Thus the EVAL command may be executed in a batch machine or in some other disconnected virtual machine.

When the EVAL command is issued, it will look for a file with file name the same as the model name and file type RQ2COMP. If it finds such a file on any accessed minidisk, it will assume that the first such file in the search order was generated by the SETUP command and solve the model defined by that file. If EVAL does not find such a file, it will terminate with an error message.

The EVAL command will next look for a file with file name the same as the model name and file type RQ2RPLY. If it finds such a file on any accessed minidisk, it will assume that the first such file in the search order is a list of replies to be used for prompts to be given by the EVAL command. The lines of the file are placed on the CMS stack.

The EVAL command may be given additional arguments after the model name. These additional arguments are also placed on the CMS stack, one per line. The arguments are stacked after any lines stacked from the RQ2RPLY file. If any argument is the word "null" an empty (blank) line is stacked for that argument. Because of the tokenizing of arguments with CMS, the arguments may not contain punctuation, i.e., the arguments should be restricted

to numeric values, the codes for the "WHAT:" prompt, "yes" and "no". Note that these restrictions do not apply to the RQ2RPLY file.

EVAL examines only the first 120 characters of a physical line. EVAL recognizes the concatenation symbol "++" as does the SETUP command (Section 2). However, the concatenation is only allowed for replies to prompts for parameter values and to the "WHAT:" prompt. Other prompts given by EVAL in regard to run continuation require only a few characters in reply, so concatenation is not considered for these prompts.

EVAL allows comments, enclosed by "/*" and "*/", in all replies except those given as arguments in the EVAL command. (This restriction is because of CMS tokenizing of arguments.) Comments are primarily useful in RQ2RPLY files.

After the EVAL command is issued, it immediately types the line "RESQ2 EXPANSION AND SOLUTION PROGRAM." If the model name has not been given on the command line, then the prompt "MODEL:" will be given, with the name expected as the reply. There are two basic phases of the EVAL command, macro expansion of submodel invocations and model solution (e.g., simulation). After the initial typed line, and the "MODEL:" prompt, if necessary, there is a noticeable delay while the module which performs the expansion is loaded into memory. After the module is loaded, it types a line giving the date of creation of the module and the current time and date. If the model has numeric and/or distribution parameters, then there will be prompts for parameter values. Each prompt consists of the parameter name followed by a colon (":"). The prompts are given in the order that the parameters are declared in the model definition. The expressions given as replies to the prompts follow the rules in Appendix 3 but are constrained to use only numeric constants, basic arithmetic operations and numeric function calls. In the case of distribution parameters, RESQ distribution functions may also be used. In the case of array parameters, all values are given on a single logical line. If fewer values are given than the number of array elements, the last value is used for the remaining elements. If more values are given than the number of array elements, the extra values are ignored.

13.1.1. Solution Summaries

With numerical solution, the same module handles submodel expansion and model solution. With simulation, after definition of parameter values, if any, the expansion module writes a temporary file to be read by the solution module, the solution module is loaded, the temporary file is read and the solution is performed. Unless simulation trace has been specified and/or the print function (Appendix 3) has been used, there will be no more typed output until the end of the solution. If numerical solution is used, the only typed output before the "WHAT:" prompts will be either an error message or the "NO ERRORS DETECTED DURING NUMERICAL SOLUTION" message. If simulation is used, the form of the lines prior to the "WHAT:" prompts depends on whether a confidence interval method has been used, and if so, which method. If the regenerative method or the spectral method is used, the form of these lines will also depend on whether or not sequential stopping was used. Several of these cases are illustrated in the examples of Section 1 and Appendix 1, as well as the examples we give here. After the solution summary has been given initially, the user may have it repeated by replying "sim" (for "simulation summary") to a "WHAT:" prompt. The solution summary is placed on the RQ2PRNT file, the EVAL command transcript file, as well as the terminal. (This file has file name the same as the model name and file type RQ2PRNT.)

Simulation without confidence intervals. If no confidence interval method has been used, then the next typed line will be "RUN END:" followed by the limit or limits which were reached and caused the run to end. Then there will either be an error message or the line

"NO ERRORS DETECTED DURING SIMULATION." If an initial portion of the run was discarded, this line will also indicate the number of discarded events. The next three lines will give the simulated time (excluding any discarded portion of the run), the CPU time consumed by the run (in seconds) and the number of simulated events (excluding any discarded portion of the run). For example, we might have

```
RUN END: MEMORY DEPARTURE LIMIT
NO ERRORS DETECTED DURING SIMULATION.  3418 DISCARDED EVENTS

                SIMULATED TIME:   812.77954
                      CPU TIME:       19.78
             NUMBER OF EVENTS:        30857
```

Independent replications. If independent replications are used, then there will be a typed line for each replication indicating the limit or limits which were reached and caused the replication to end. Then there will either be an error message or the line "NO ERRORS DETECTED DURING SIMULATION." If initial portions of the replications were discarded, this line will also indicate the number of discarded events. (If the run ends in the midst of a replication other than the first because of the CPU limit, the number of events for the partially completed replication will be included in the discarded event count. However, if the run is continued, this replication will resume where it stopped and the events recovered will be removed from the discarded event count.) The next four lines will give the mean simulated time per replication (excluding discarded portions), the total CPU time consumed by the run (in seconds), the mean number of simulated events per replication (excluding discarded portions) and the number of replications. For example, we might have

```
REPLICATION   1: SET_TOTAL DEPARTURE LIMIT
REPLICATION   2: SET_TOTAL DEPARTURE LIMIT
REPLICATION   3: SET_TOTAL DEPARTURE LIMIT
REPLICATION   4: SET_TOTAL DEPARTURE LIMIT
REPLICATION   5: SET_TOTAL DEPARTURE LIMIT
NO ERRORS DETECTED DURING SIMULATION.  19779 DISCARDED EVENTS

  SIMULATED TIME PER REPLICATION:      207.36981
                        CPU TIME:         260.54
NUMBER OF EVENTS PER REPLICATION:          35791
          NUMBER OF REPLICATIONS:              5
```

If independent replications are used but not even the first replication is completed, e.g., the CPU limit is reached before the first replication ends normally, then the output will be essentially the same as simulation without confidence intervals except that the number of replications will be given as zero (0). Assuming the first replication did not complete because of the CPU limit and not because of an error, the run continuation dialogue may be used to increase the CPU limit and continue the run where it stopped.

Regenerative method without sequential stopping. If the regenerative method is used without the sequential stopping rule, then the next typed line will be "RUN END:" followed by the guidelines and/or limit (CPU) which were reached. Then there will either be an error message or the line "NO ERRORS DETECTED DURING SIMULATION." If part of the run was discarded because the simulation did not begin in the regeneration state and/or the simulation did not end in the regeneration state (because of an error or the CPU limit), this line will indicate the number of discarded events. (If the run ends in the midst of a regeneration cycle other than the first because of the CPU limit, the number of events for the partially completed cycle will be included in the discarded event count. However, if the run is contin-

ued, this cycle will resume where it stopped and the events recovered will be removed from the discarded event count.) The next four lines will give the simulated time (excluding any discarded portion of the run), the CPU time consumed by the run (in seconds), the number of simulated events (excluding any discarded portion of the run) and the number of regeneration cycles. For example, we might have

```
RUN END: EVENT GUIDELINE MEMORY DEPARTURE GUIDELINE CPU LIMIT
NO ERRORS DETECTED DURING SIMULATION.  3418 DISCARDED EVENTS

                SIMULATED TIME:   812.77954
                      CPU TIME:       19.78
            NUMBER OF EVENTS:         30857
            NUMBER OF CYCLES:            27
```

If fewer than two regeneration cycles were completed, confidence intervals will not be available and run continuation will not be allowed.

Regenerative method with sequential stopping. If the regenerative method is used with the sequential stopping rule, then for each normally completed sampling period there will be a line "SAMPLING PERIOD END:" followed by the guidelines which caused the sampling period to end. If the last sampling period does not end because of its guidelines but because of the CPU limit or an error, then the next typed line will be "RUN END:" followed by the guidelines and/or limit (CPU) which were reached. Then there will either be an error message or the line "NO ERRORS DETECTED DURING SIMULATION." If part of the run was discarded because the simulation did not begin in the regeneration state and/or the simulation did not end in the regeneration state (because of an error or the CPU limit), this line will indicate the number of discarded events. (If the run ends in the midst of a regeneration cycle other than the first because of the CPU limit, the number of events for the partially completed cycle will be included in the discarded event count. However, if the run is continued, this cycle will resume where it stopped and the events recovered will be removed from the discarded event count.) The next four lines will give the simulated time (excluding any discarded portion of the run), the CPU time consumed by the run (in seconds), the number of simulated events (excluding any discarded portion of the run) and the number of regeneration cycles. For example, we might have

```
SAMPLING PERIOD END: MEMORY DEPARTURE GUIDELINE
RUN END: CPU LIMIT
NO ERRORS DETECTED DURING SIMULATION.  3418 DISCARDED EVENTS

                SIMULATED TIME:   812.77954
                      CPU TIME:       19.78
            NUMBER OF EVENTS:         30857
            NUMBER OF CYCLES:            27
```

Spectral method without sequential stopping. If the spectral method is used without the sequential stopping rule, then the next typed line will be "RUN END:" followed by the limits which were reached. Then there will either be an error message or the line "NO ERRORS DETECTED DURING SIMULATION." If an initial portion of the run was discarded, this line will indicate the number of discarded events. The next three lines will give the simulated time (excluding any discarded portion of the run), the CPU time consumed by the run (in seconds) and the number of simulated events (excluding any discarded portion of the run). For example, we might have

```
RUN END: EVENT LIMIT
NO ERRORS DETECTED DURING SIMULATION.  3418 DISCARDED EVENTS

                SIMULATED TIME:    812.77954
                      CPU TIME:        19.78
                NUMBER OF EVENTS:      30857
```

Spectral method with sequential stopping. If the spectral method is used with the sequential stopping rule, then for each normally completed sampling period there will be a line "SAMPLING PERIOD END:" followed by the limits which caused the sampling period to end. If the last sampling period does not end because of its limits but because of the CPU limit or an error, then the next typed line will be "RUN END:" followed by the limits which were reached. Then there will either be an error message or the line "NO ERRORS DETECTED DURING SIMULATION." If an initial portion of the run was discarded, this line will indicate the number of discarded events. The next three lines will give the simulated time (excluding any discarded portion of the run), the CPU time consumed by the run (in seconds) and the number of simulated events (excluding any discarded portion of the run). For example, we might have

```
SAMPLING PERIOD END: MEMORY DEPARTURE LIMIT
RUN END: CPU LIMIT
NO ERRORS DETECTED DURING SIMULATION.  3418 DISCARDED EVENTS

                SIMULATED TIME:    812.77954
                      CPU TIME:        19.78
                NUMBER OF EVENTS:      30857
```

13.1.2. Performance Measures

After the solution summary, the user is prompted with "WHAT:", meaning "What performance measures do you want to see?" The replies to "WHAT:" are codes indicating performance measures. The code "all" indicates "all of the usual measures;" the measures included and excluded in "all" will be indicated below. The performance measures are placed on the RQ2PRNT file, the EVAL command transcript file, as well as the terminal. (This file has file name the same as the model name and file type RQ2PRNT.) After the performance measures are shown, the "WHAT:" prompt will be repeated until a null reply is given.

Suffixes may be added to the performance measure codes to control presentation of confidence intervals and/or to control the elements (e.g., queues) for which the measures will be given. The confidence interval suffix, if any, precedes the suffix for control of elements considered. Without a confidence interval suffix, only point estimates are given. The two confidence interval suffixes are "ci", which indicates that confidence intervals are to be given instead of point estimates, and "bo" (for "both") which indicates that both point estimates and confidence intervals are to be given. For example, "all" results in only point estimates, "allci" results in only confidence intervals and "allbo" results in point estimates and all available confidence intervals for the "usual" performance measures for all queues and nodes.

A suffix for control of elements considered consists of either "(*)" or a parenthesized list of names of elements (e.g., queues). There are slightly different conventions for "all" and the other codes, e.g., "ut" for utilization. With codes other than "all", e.g., "ut", results for nodes belonging to a queue are not given if the code is given without a suffix indicating the node results are to be given as well as the queue results; the code without a suffix results in measures for all queues, and, if appropriate to the measure, results for nodes not associated with queues. The "(*)" suffix indicates that measures for nodes associated with a queue are

to be given as well as the queue measures, for all queues. For example, if a queue has two classes, then "ut" will give only the utilization for the queue overall, while "ut(*)" will give the class specific utilizations. With "all" results for all queues and nodes are given unless there is a suffix giving a list of names of elements. The "(*)" suffix has no effect with "all". With "all" and with the other codes, if there is a suffix giving a list of names of elements, e.g., "all(cssm1.cpuq,cssm1.setcmdtype)" or "ut(line.msg_in,line.cnt_in), then only measures for those elements will be given.

We now list the individual codes and the definitions of the associated performance measures. First we list the "usual" measures included in "all", in the order listed with "all", then we list the other codes and their meanings. *In all cases, measures are only available for a queue or node if there has been at least one departure from that queue or node.*

ut – utilization. For an active queue the utilization is defined as the fraction of time a server is in use. (For an infinite server queue the utilization of each server is zero.) For a class the utilization is defined as the fraction of time a server is in use by jobs of that class. In both cases, these are average values over all servers of the queue. If a queue has heterogeneous servers (because of different rates and/or different classes accepted) then utilizations will also be given for each server that was used during the simulation. For a passive queue the utilization is defined as the fraction of time a token is in use. For an allocate node the utilization is defined as the fraction of time a token is in use by jobs of that node. In both cases, these are average values over all tokens of the queue. (Tokens are always homogeneous.) *If the number of tokens is not constant, because of use of create and/or destroy nodes, utilization may not be well defined.* See the discussion of the "tu" (mean tokens in use) code. (If the number of tokens at the end of simulation is not the same as the number at the beginning of simulation, a utilization will not be reported.)

tp – throughput. Throughput is defined as the average number of departures per unit time. For active queues and classes, departures correspond to service completions. For passive queues, "plain" allocate nodes and AND allocate nodes, departures correspond to release or destruction of tokens. For OR allocate nodes, departures correspond to release of tokens, destruction of tokens or the termination of a request for tokens which has been satisfied by another queue. Note that an AND or OR allocate node will have separate performance measures for each queue to which it belongs. Except when measures are requested for an AND or OR allocate node separately, these measures will be grouped with the corresponding queues. Departures for split nodes and fission nodes consider only the entering job, not the jobs generated.

ql – mean queue length. Queue length for active queues and classes is defined as the number of jobs waiting for or holding servers. For passive queues, "plain" allocate nodes, AND allocate nodes and OR allocate nodes, queue length is defined as the number of jobs waiting for or holding tokens. For passive queues with both release and destroy nodes, a "Little's Rule" estimate of the mean queue length associated with release and destruction of tokens, respectively, is obtained from the throughput multiplied by the mean queueing time. Note that these two mean queue lengths may not add up to the value reported for the queue

because of jobs still waiting for tokens at the end of simulation and/or jobs still holding tokens at the end of simulation.

sdql – standard deviation of queue length.

qt – mean queueing time. Queueing time for active queues and classes is defined as the time spent waiting for or holding servers. For passive queues, "plain" allocate nodes and AND allocate nodes, queueing time is defined as the time spent waiting for or holding tokens. For OR allocate nodes, for the queue which provides tokens, queueing time is defined as the time spent waiting for or holding tokens. For OR allocate nodes, for a queue which does not provide tokens, queueing time is defined as the time spent waiting for tokens. For passive queues with both release and destroy nodes, queueing times are categorized into those ending with release of tokens (or end of waiting at an OR allocate node) and those ending with destruction of tokens. For all of these cases, except for models using the regenerative method, only completed queueing times are considered in the mean queueing time and other queueing time measures. For the regenerative method only, mean queueing time is not computed directly but is computed by a "Little's Rule" argument so that queueing times in progress may be allowed and still have rigorous computation of confidence intervals.

sdqt – standard deviation of queueing time.

tu – mean tokens in use. This applies only to passive queues. The number of tokens in use is the number of tokens allocated to jobs. If the number of tokens of a queue is constant, then the mean number of tokens in use is equal to the number of tokens multiplied by the utilization. If the number of tokens of a queue fluctuates, because of the use of create and destroy nodes, the number of tokens in use is well defined even though the utilization is not well defined.

tt – mean total tokens in pool. This applies only to passive queues. The total number of tokens is constant (and equal to the number given on the "TOKENS:" line) unless create and/or destroy nodes are used.

qld – queue length distribution. This only applies if the dialogue specifies gathering of queue length distributions, and then only to the queues and nodes specified in the dialogue and only up to the maximum lengths specified in the dialogue. The probabilities of all queue lengths with non-zero probabilities are given.

qtd – queueing time distribution. This only applies if the dialogue specifies gathering of queueing time distributions, and then only to the queues and nodes specified in the dialogue and only for the values specified in the dialogue. The cumulative probabilities of queueing time being less than or equal to each specified value are given.

tud – distribution of tokens in use. This applies only to passive queues. This only applies if the dialogue specifies gathering of token use distributions, and then only to the queues specified in the dialogue and only up to the maximum values specified in the dialogue. The probabilities of all numbers of tokens in use with non-zero probabilities are given.

ttd – distribution of total tokens in pool. This applies only to passive queues. This only applies if the dialogue specifies gathering of total token distributions, and then only to the queues specified in the dialogue and only up to the maximum values specified in the dialogue. The probabilities of all numbers of tokens with non-zero probabilities are given.

mxql – maximum queue length.

mxqt – maximum queueing time.

po – open chain population. This applies only to open chains. The population is the number of jobs in the chain. This measure gives the mean number of jobs in the chain.

rtm – open chain response time. This applies only to open chains. The response time is the time between a job's entering the chain, either from a source or split node, and a job's departure through a sink. The response time is estimated by a "Little's Rule" argument. The chain throughput is defined as the mean number of jobs of the chain which depart (through the sink) per unit time. The mean response time is determined by dividing the chain population by the chain throughput. Thus the mean response time is inflated by the jobs still in the chain.

All of the above measures are included in "all". None of the values below are included in "all".

nd – number of departures. This is defined as discussed in the definition of throughput.

st – mean service time. This applies only to active queues and classes. The user has specified a distribution, including the mean of that distribution, but statistical variability will usually result in a slightly different mean, which is the value reported for this code. Only completed service times are considered, except for the regenerative method. With the regenerative method only, mean service time is determined indirectly from the utilization divided by the throughput.

The following values are not truly performance measures in the sense of the above. In particular, the values reported with the following codes are those at the current state of the simulated network, even if the simulation is in the midst of an incomplete replication or regeneration cycle which is ignored for the above measures.

lng – final lengths. This gives the queue lengths at the end of simulation.

jv – final job variable values for jobs still in the network. The jobs currently in the network are listed by queue (or node, if "(*)" or an explicit list of nodes is used) in the order found in the queue. The internal number of the job is given and the values for each job variable are given.

cv – final CV values. For each chain, the values of the chain variables are listed.

gv – final values of global variables. The final values of global variables are listed.

13.1.3. Run Continuation and Multiple Solutions

After a null reply to a "WHAT:" prompt, if run continuation is allowed the next line will be "CONTINUE RUN:", which requires a "yes" or "no" reply. If the reply is "no" or run continuation is not allowed, then if the model has parameters, a new prompt for the first parameter will be given. (If there are no parameters, then the EVAL command terminates.) A new set of parameters and solution process may begin at this point, or a null reply may be given to end the EVAL command.

Run continuation is allowed provided that the simulation has not terminated because of an error, that the simulation did not terminate because of an "infinite" routing loop which consumes no simulated time, that if independent replications are used that not all replications have completed, and that if the regenerative method is used there have been at least two completed cycles. If "yes" is given to the "CONTINUE RUN:" prompt, then there will be prompts to control the run continuation. Except for models using independent replications or sequential stopping, these prompts will be for new values for limits or guidelines which do not already have "infinite" values. New values are required for limits or guidelines which have been reached. New values for the other limits and guidelines are optional. Limits may only be increased or left the same by giving a null reply. With independent replications only the CPU limit may be increased. With sequential stopping, only the extra sampling period and CPU limit values may be increased.

When a run is continued, the "RUN END:" and/or "SAMPLING PERIOD END:" lines from earlier portions of the run will be repeated in the simulation summary for later portions of the run. Otherwise the run is the same as if the larger limits had been specified initially (with an appropriately smaller initial portion discarded, if applicable). A run may be continued several times, if appropriate. When a run is finally terminated, new runs with new parameter values may be made if the model has parameters.

13.2. EVALT Command

In most respects the EVALT command is the same as the EVAL command. The EVALT command is intended for use only when the user is providing a USER numeric function, as discussed in Appendix 3. Rather than using the simulation module, which already has the default (error stop) copy of USER, EVALT runs the simulation from the object code libraries and object code files found on accessed mini-disks. If the accessed mini-disks contain any files with file name the same as the name of an internal simulation procedure, e.g., USER, and file type TEXT, then these files will be used instead of the standard copies of those procedures. Thus the user should be sure to have a file USER TEXT on an accessed mini-disk and to avoid having other TEXT files which might be used inappropriately by the EVALT command. Other than these characteristics, the only other noticeable differences between the EVAL and EVALT commands are that EVALT is slightly slower to begin simulation, because of the time required to link the entry points together, and that there will be an additional line, "EXECUTION BEGINS...", when the simulation begins.

13.3. EVAL Command Files

We have already discussed or mentioned most of the files used or produced by the EVAL command. The normal input to the EVAL command is from three files: (1) SYSIN - the EVAL EXEC issues a CMS FILEDEF command defining SYSIN to be the terminal. (2) The reply file (RQ2RPLY) if one exists and (3) the model definition file (RQ2COMP) produced by SETUP. In addition, EVAL will use as input either RESQ2 APLMBD, which is used to

define the sizes of certain internal tables for the simulation, or RESQ2 NUMERD, which constrains the number of queue-dependent queues allowed in a network to be solved numerically.

EVAL cannot determine in advance the maximum size of the simulation event list or the maximum numbers of jobs and job copies in the network during simulation. File RESQ2 APLMBD on the mini-disk containing EVAL EXEC contains sizes for these tables and buffers. The default content of the file is

```
MAXEL=256, MAXJL=1024, MAXJDL=256;
```

where MAXEL is the maximum size of the event list, MAXJL is the maximum number of jobs plus job copies, and MAXJDL is the maximum number of jobs not counting job copies. The user may have a copy of RESQ2 APLMBD on a mini-disk in the search order before the mini-disk containing the EVAL EXEC, to be used instead of the default copy. The user may increase (or decrease within reason) these sizes in this copy of RESQ2 APLMBD. If an error message says that event list, job list or job data list storage has been exceeded, then MAXEL, MAXJL or MAXJDL, respectively, should be increased for that model. (This assumes that the model is not "running wild," e.g., that jobs are not just accumulating at some node.) On the other hand, if the user wishes to reduce the virtual storage required, many models will run with smaller values, e.g., many models will run with

```
MAXEL=32, MAXJL=64, MAXJDL=32;
```

Each event list element requires 32 bytes of storage, i.e., with MAXEL=256 the event list elements take 8192 bytes of storage. The list of jobs and job copies (MAXJL) takes 56 bytes per element. The job data (MAXJDL) storage depends on the number of job variables. The storage required per element is 56 bytes plus 8 bytes per job variable, e.g., if the default maximum job variable index of one is used, the storage per element is 72 bytes.

Since the numerical solution becomes increasingly expensive as the number of queues with queue length dependent service rates increases, the file RESQ2 NUMERD contains a limit to the number of queue length dependent queues allowed. The default content of the file is

```
MVAQDL=4;
```

The user is free to create a copy of RESQ2 NUMERD earlier in the search order to set this to any non-negative limit.

While executing, the EVAL command produces three files: (1) SYSPRINT - the EVAL EXEC issues a CMS FILEDEF command defining the terminal to be SYSPRINT. (2) RQ2PRNT - the transcript of the terminal interaction, e.g., for printing, and (3) RQ2NTWK - this is a temporary file which is written by the EVAL command and later erased by the EVAL command.

Figure 13.1 shows these files and their relationships with the commands.

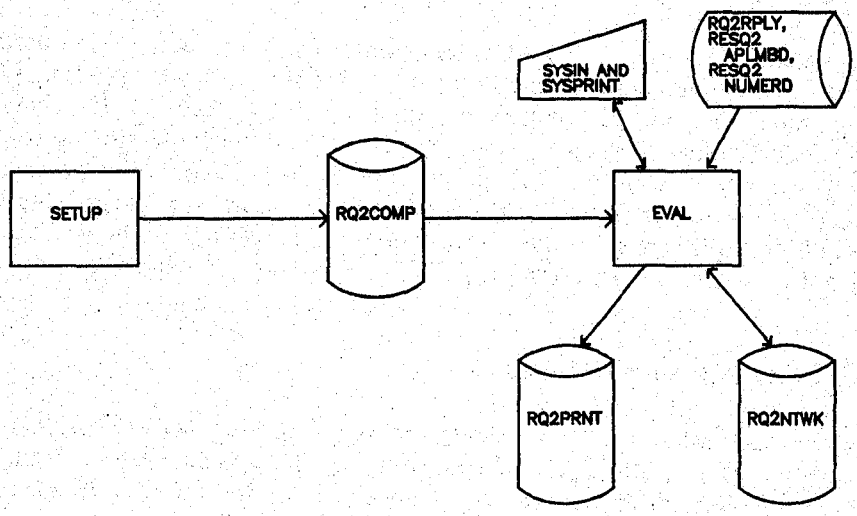

Figure 13.1 - Files used with EVAL

14. PL/I EMBEDDING

Instead of using the EVAL or EVALT commands after a model has been defined with the SETUP command, model expansion may be embedded within a PL/I program. (This assumes that the PL/I optimizing compiler is available independent of RESQ.) This may be done in order (1) to produce tables or graphs of results, (2) to coordinate solution of several separate models in a hierarchical solution, (3) to provide a preprocessor for determining model parameters and/or (4) to provide a postprocessor for manipulating model solutions prior to display. Section 14.1 discusses the basic procedures for PL/I embedding and the interface to CMS. Section 14.2 discusses procedures for plotting graphs of model results.

14.1. Basic Procedures and CMS Commands

The basic steps in using PL/I embedding are (1) to produce an RQ2COMP file using the SETUP command, (2) to produce a PL/I program, (3) to compile that program, (4) to issue CMS commands for file definition and object library definition and (5) to execute the user's PL/I program. Several different orderings of these steps are possible, but we will assume assume the RQ2COMP file has been produced and discuss the remaining steps in the order just listed.

14.1.1. The PL/I Program

The PL/I program calls procedures provided by RESQ to (1) establish the model definition(s) given by the RQ2COMP file(s), (2) to specify model parameters, (3) to perform the model expansion and solution, and (4) to determine the results of model solution. The name of the program should not be a name used in EXPANSUB TXTLIB, APLOMB2 TXTLIB or MVASUB TXTLIB. (See Section 14.1.4 for description of these libraries.) Normally the source file for the program will have file name the same as the procedure name and file type either PLIOPT or PLI.

The RESQ procedure READMD reads a model definition file (RQ2COMP) which has been produced by the SETUP command. READMD has no parameters, so the declaration

```
DECLARE READMD ENTRY;
```

and calling statement

```
CALL READMD;
```

are sufficient. Normally the file read will have data set name (in the OS sense) RSQ2IP, to be used in the CMS FILEDEF statements. However, if several different RQ2COMP files are to be read by the same PL/I program, the TITLE option of the PL/I OPEN statement may be used to define other data set names, e.g.,

```
OPEN FILE(RSQ2IP) TITLE('MODEL1');
CALL READMD;
...
   /*Define parameters, solve, obtain results for MODEL1*/
...
OPEN FILE(RSQ2IP) TITLE('MODEL2');
CALL READMD;
...
```

```
    /*Define parameters, solve, obtain results for MODEL2*/
...
```

After the call to READMD, all parameter values need to be defined before calling a procedure to expand and solve the model. Only scalar numeric and vector numeric parameters are allowed in models to be solved by PL/I embedding. Once a parameter value has been defined by one of the following two procedures, its value need not be defined again unless or until READMD is called again, i.e., if a model has several parameters, expansion and solution may be performed several times, changing some parameters and leaving the existing values of other parameters intact without explicitly resetting parameters to their current values. The RESQ procedure STPARM is used to define values for scalar parameters, one at a time. The declaration for STPARM should be of the form

```
DECLARE STPARM ENTRY (CHAR(10),FLOAT BIN(21));
```

where the first STPARM parameter gives the name of the model parameter and the second STPARM parameter gives the model parameter value, e.g.,

```
CALL STPARM('THINKTIME',5.2);
```

Values for vector numeric parameters are defined by calls to RESQ procedure STPRMV, one vector at a time. The declaration for STPRMV should be of the form

```
DECLARE STPRMV ENTRY(CHAR(10),(*) FLOAT BIN(21));
```

where the first STPRMV parameter is the name of the model parameter and the second STPRMV parameter is a vector of values for the model parameter, e.g.,

```
CALL STPRVM('VRATES',RATES);
```

where RATES is declared by

```
DECLARE RATES(5) FLOAT BIN(21);
```

Model expansion and solution are performed by RESQ2A for simulation and RESQ2M for numerical solution. The entry declaration for either of these procedures is

```
DECLARE RESQ2x ENTRY(FIXED BIN(31));
```

where "x" is either "A" for simulation or "M" for numerical solution. The parameter for RESQ2A and RESQ2M indicates whether the dialogue giving the solution summary, "WHAT:" prompts for performance measures and run continuation, is to be entered at the end of solution. If the parameter is non-zero, e.g.,

```
CALL RESQ2A(1);
```

then the dialogue is entered, and if the parameter is zero, the dialogue is not entered (and run continuation is not possible).

Three procedures are available to obtain solution results after calling procedure RESQ2A or RESQ2M. RESQ procedure TYPEVL can be used to enter the dialogue for solution summary and "WHAT:" prompts for performance measures, but run continuation is not possible and the "lng", "jv", "cv" and "gv" codes may not be used in reply to "WHAT:" prompts. TYPEVL has no parameters, so

```
DECLARE TYPEVL ENTRY;
```

and

```
CALL TYPEVL;
```

are sufficient. RESQ procedure FNLMSG may be used to obtain the "final message" produced by the solution, i.e., either the "NO ERRORS ..." message or an error message. FNLMSG has a fixed 80 character string as its parameter, e.g.,

```
DECLARE FNLMSG ENTRY(CHAR(80)),
        FMSG   CHAR(80);
```

and

```
CALL FNLMSG(FMSG);
```

could be used to place the final message in FMSG. RESQ procedure GTRSLT will retrieve a specified performance measure for a given element. The declaration is of the form

```
DECLARE GTRSLT ENTRY (CHAR(*) VARYING,
                      CHAR(*) VARYING,(3) FLOAT BIN(21));
```

where the first parameter is the name of the element (possibly including a parenthesized array index), the second parameter is a code used in reply to "WHAT:" (excluding suffixes) and the third parameter is used for the point estimate and the confidence interval, if available, e.g., after

```
DECLARE OP(3) FLOAT BIN(21);
CALL GTRSLT('Q2','QL',OP);
```

the mean queue length for "Q2" would be given in OP(1) and, if available, a confidence interval for the mean queue length would be given in OP(2) and OP(3), with the lower value in OP(2). (If no confidence interval is available, OP(2) and OP(3) will be −1.) The element name can be the name of any queue or node in the model for which the specified performance measure exists. Only codes "ut", "tp", "ql", "sdql", "qt" and "sdqt" may be used.

14.1.2. PL/I Compilation

Normally the source file for the program will have file name the same as the procedure name and file type either PLIOPT or PLI. The CMS PLIOPT command is used to compile the program, e.g.,

```
PLIOPT myprog
```

could be used to compile program "myprog" and produce file MYPROG TEXT for use in the LOAD command as discussed in Section 14.1.4.

14.1.3. CMS Commands for Execution

Prior to execution of the program, the CMS GLOBAL and FILEDEF commands must be used to establish the proper environment. The GLOBAL command is used to identify the

TXTLIB's (object code libraries) to be used and the search order of these libraries, e.g., the statement

```
GLOBAL TXTLIB EXPANSUB MVASUB APLOMB2 PLILIB
```

declares that EXPANSUB TXTLIB will be the first library searched for external references, MVASUB TXTLIB will be the second library searched, etc. EXPANSUB TXTLIB contains the procedures described in Section 14.1.1 and other procedures for model expansion. MVASUB TXTLIB contains the procedures for numerical solution and APLOMB2 TXTLIB contains the procedures for simulation. It is assumed that the PL/I optimizing compiler run time library is available as PLILIB TXTLIB.

The CMS FILEDEF command is used to associate the data set names used in the PL/I procedures with files in the CMS environment, e.g., the terminal, files on mini-disks and virtual spool files. The FILEDEF command must be used for data set names SYSPRINT, RSQ2RS, APLMBD (if RESQ2A is to be called), NUMERD (if RESQ2M is to be called), and either RSQ2IP or corresponding data set names given with the TITLE option of the PL/I OPEN statement as discussed in Section 14.1.1. Assuming the model name is "mymodel" and the TITLE option is not used, the following FILEDEF statements are recommended (and could be placed in a user written EXEC file).

```
FILEDEF SYSPRINT TERMINAL (PERM LRECL 132 BLKSIZE 132 RECFM F
FILEDEF RSQ2RS DISK mymodel RQ2PRNT A (PERM RECFM V BLKSIZE 141
FILEDEF RQ2PLOT DISK mymodel RQ2PLOT A (PERM RECFM V BLKSIZE 141
FILEDEF APLMBD DISK RESQ2 APLMBD * (PERM RECFM F BLKSIZE 80
FILEDEF NUMERD DISK RESQ2 NUMERD * (PERM RECFM F BLKSIZE 80
FILEDEF RSQ2IP DISK mymodel RQ2COMP * (PERM RECFM V BLKSIZE 2500
```

After the GLOBAL and FILEDEF statements have been issued, the LOAD and START commands are used to execute the program, e.g., if the main program has name "myprog", the following could be used

```
LOAD myprog (NODUP RESET DMSIBM
START DMSIBM ISASIZE(-100K)
```

The RPLOT EXEC discussed in the following section may also be used where plots are not desired (without changes to the EXEC).

14.2. Plotting Procedures

Several procedures are supplied with RESQ for producing low resolution graphs of model results on a terminal, line printer or other appropriate character oriented device. Other PL/I callable graphics packages supplied by the user may be used in a similar manner.

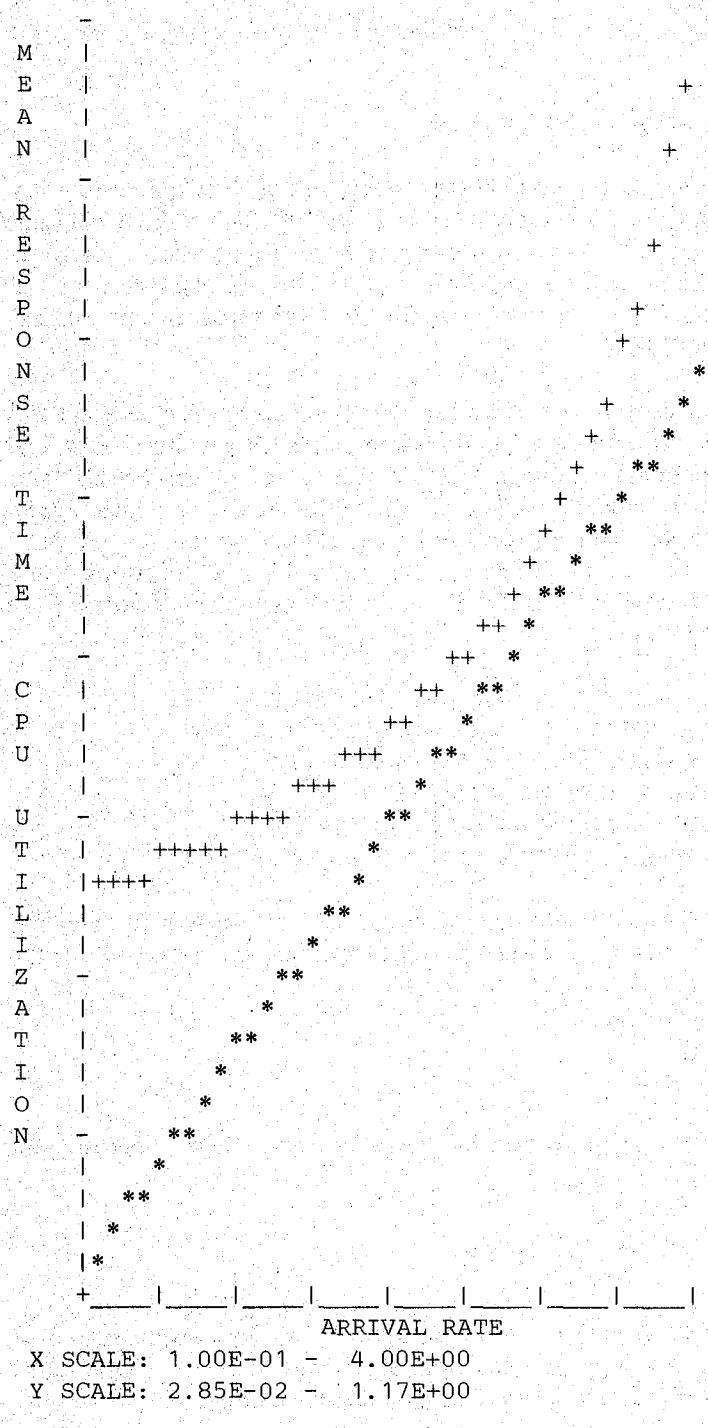

Figure 14.1 - Example Graph of Model Results

The following declaration could be used for the plotting procedures

```
DECLARE
   RQSET  ENTRY(FIXED BIN(31),FIXED BIN(31)),
   RQPLOT ENTRY((*,*) FLOAT BIN(21)),
   RQXLBL ENTRY(CHAR(*) VARYING),
   RQYLBL ENTRY(CHAR(*) VARYING),
```

```
   RQVIEW ENTRY;
```

The RESQ procedure RQSET is used to define the size of the graph, in terms of rows and columns available for displaying curves. The first RQSET parameter is the number of rows and the second is the number of columns. Five additional rows and five additional columns are used for labeling. For example,

```
CALL RQSET(20,40);
```

defines that there are to be 20 rows and 40 columns for curves. (The entire plot will consist of 25 rows and 45 columns.) The RESQ procedure RQPLOT is given an array defining the data to be plotted. The data array must have at least two columns, for plotting a single curve, and should have an additional column for each additional curve to be plotted. The data array must have at least as many rows as the number of columns specified in the call to RQSET. The first column gives the values for the X axis, and each additional column defines Y axis values for a curve. For example, we might have

```
DECLARE
   DATA(60,3) FLOAT BIN(21);
...
   /*Define elements of data for first 40 rows*/
...
CALL RQPLOT(DATA);
```

to plot two curves. RESQ procedure RQXLBL is used to give a label for the X axis, and RESQ procedure RQYLBL is used to give a label for the Y axis, e.g.,

```
CALL RQXLBL('                ARRIVAL RATE');
CALL RQYLBL('MEAN RESPONSE TIME  CPU UTILIZATION');
```

RESQ procedure RQVIEW displays the graph on the terminal and on the file with data set name RQ2PLOT. A CMS FILEDEF statement must be used for RQ2PLOT before executing a program calling RQVIEW, e.g.,

```
FILEDEF RQ2PLOT DISK mymodel RQ2PLOT A (PERM RECFM V BLKSIZE 141
```

Following is a complete program which could be used with model EXAMP1 in Appendix 1:

```
EXAMP1: PROCEDURE OPTIONS(MAIN) REORDER;
  DECLARE
    N FIXED BIN(31),
    (T,DATA(40,3),OP(3)) FLOAT BIN(21),
    FMSG CHAR(80),
    (FLOAT,SUBSTR) BUILTIN,
  /*Entry points for RESQ routines:*/
    READMD ENTRY,
    STPARM ENTRY (CHAR(10),FLOAT BIN(21)),
    RESQ2M ENTRY(FIXED BIN(31)),
    FNLMSG ENTRY(CHAR(80)),
    GTRSLT ENTRY (CHAR(*) VARYING,
                  CHAR(*) VARYING,(3) FLOAT BIN(21)),
  /*Entry points for RESQ plotting routines:*/
    RQSET  ENTRY(FIXED BIN(31),FIXED BIN(31)),
```

```
    RQPLOT ENTRY((*,*) FLOAT BIN(21)),
    RQXLBL ENTRY(CHAR(*) VARYING),
    RQYLBL ENTRY(CHAR(*) VARYING),
    RQVIEW ENTRY;
  CALL READMD; /* Reads RQ2COMP file produced by SETUP*/
  CALL STPARM('CPIOCYCLES',8.0); /*Set parameter value*/
  DO N=1 TO 40;
    DATA(N,1)=FLOAT(N)/10.0;
    CALL STPARM('ARVL_RATE',FLOAT(N)/10.0); /*Set parameter value*/
    CALL RESQ2M(0); /* Expands model & solves numerically*/
    CALL FNLMSG(FMSG);
    IF SUBSTR(FMSG,1,9)¬='NO ERRORS' THEN
      STOP;
    CALL GTRSLT('CPUQ','QL',OP); /* Get result */
    T=OP(1);
    CALL GTRSLT('DISKQ','QL',OP); /* Get result */
    DATA(N,2)=(T+OP(1))/(FLOAT(N)/10.0); /*Mean response time
                                          (Little's Rule)    */
    CALL GTRSLT('CPUQ','UT',OP); /* Get result */
    DATA(N,3)=OP(1);
  END;
  CALL RQSET(40,40);
  CALL RQPLOT(DATA);
  CALL RQXLBL('               ARRIVAL RATE');
  CALL RQYLBL('MEAN RESPONSE TIME  CPU UTILIZATION');
  CALL RQVIEW;
END;
```

After compiling this procedure, with the PLIOPT command, we could use the RPLOT EXEC, e.g.,

```
rplot examp1 examp1
```

to get the plot shown in Figure 14.1. Following is a listing of RPLOT EXEC:

```
&CONTROL OFF
&IF &INDEX = 2 &SKIP 4
&BEGTYPE
RPLOT REQUIRES EXACTLY TWO ARGUMENTS, MODEL NAME AND PROGRAM NAME
&END
&EXIT 100
STATE &1 RQ2COMP *
&IF &RETCODE = 0 &SKIP 2
&TYPE &1 RQ2COMP FILE NOT FOUND.  USE SETUP FIRST.
&EXIT 28
GLOBAL TXTLIB APLOMB2 EXPANSUB MVASUB PLILIB
FILEDEF SYSPRINT TERMINAL (PERM LRECL 132 BLKSIZE 132 RECFM F
FILEDEF RSQ2RS DISK &1 RQ2PRNT A (PERM RECFM V BLKSIZE 141
FILEDEF RQ2PLOT DISK &1 RQ2PLOT A (PERM RECFM V BLKSIZE 141
FILEDEF RSQ2IP DISK &1 RQ2COMP * (PERM RECFM V BLKSIZE 2500
FILEDEF APLMBD DISK RESQ2 APLMBD * (PERM RECFM F BLKSIZE 80
FILEDEF NUMERD DISK RESQ2 NUMERD * (PERM RECFM F BLKSIZE 80
LOAD &2 (NODUP NOMAP RESET DMSIBM
START DMSIBM ISASIZE(-100K)
```

APPENDIX 1 - ADDITIONAL EXAMPLES

This appendix discusses three complete examples which illustrate aspects of RESQ not featured in the example of Section 1. The first example is a very simple model with two queues in an open chain. This example is solved numerically. The second example is related to the example of Section 1, but contains a more detailed representation of an I/O subsystem, including effects of channel and device interaction, and represents round robin scheduling at the processor. (Terminals and memory are ignored in this second example.) The third example shows how passive queues, split nodes, fission nodes, fusion nodes and other RESQ elements may be used to simply represent protocols in communication systems.

A1.1. Numerically Solved Model

Extremely simple queueing models are often sufficient to make initial system design decisions, e.g., to reject designs with substantially poorer performance than other designs being considered. Cyclic queueing networks representing only CPU and disks have been found useful in a number of applications. Figure A1.1 shows such a model of a transaction driven computing system. Transactions arrive at the CPU for processing and then alternate CPU and disk activity until the transaction is completed. The disks are represented by a single queue in the model.

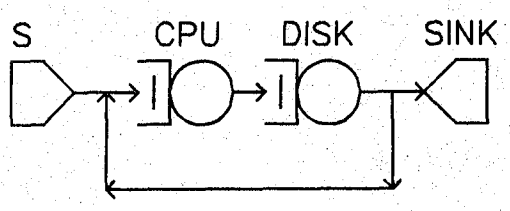

Figure A1.1 - Open Chain Cyclic Queue Model

Following is a possible dialogue file for definition of the model:

```
MODEL:examp1
   METHOD:numerical
   NUMERIC PARAMETERS:arvl_rate cpiocycles
   QUEUE:cpuq
      TYPE:ps
      CLASS LIST:cpu
         SERVICE TIMES:standard(.025,5)
   QUEUE:diskq
      TYPE:active
      SERVERS:2
      DSPL:fcfs
      CLASS LIST:disk
         WORK DEMANDS:.019
   CHAIN:trnsactns
      TYPE:open
      SOURCE LIST:s
      ARRIVAL TIMES:1/arvl_rate
```

```
      :s->cpu->disk->sink cpu;1/cpiocycles 1-1/cpiocycles
END
```

The arrival rate of transactions and the mean number of CPU-I/O cycles per transaction are left as parameters to be defined when the model is solved. The CPU is represented as having processor sharing scheduling and hyperexponential service times with mean 25 milliseconds and coefficient of variation 5 (i.e., standard deviation 125 milliseconds). Two disks are represented by a single queue, with the service times assumed to be exponential with mean 19 milliseconds. Following is the RQ2PRNT file obtained for two sets of parameter values:

```
RESQ2 VERSION DATE: MARCH 3, 1982 -  TIME: 21:29:10  DATE: 03/09/82
MODEL:EXAMP1
ARVL_RATE:3
CPIOCYCLES:8
NO ERRORS DETECTED DURING NUMERICAL SOLUTION.

WHAT:all


ELEMENT         UTILIZATION
CPUQ            0.60000
DISKQ           0.22800


ELEMENT         THROUGHPUT
CPUQ            24.00000
DISKQ           24.00000


ELEMENT         MEAN QUEUE LENGTH
CPUQ            1.50000
DISKQ           0.48100


ELEMENT         MEAN QUEUEING TIME
CPUQ            0.06250
DISKQ           0.02004


ELEMENT         OPEN CHAIN POPULATION
TRNSACTNS       1.98100


ELEMENT         OPEN CHAIN RESPONSE TIME
TRNSACTNS       0.66033

WHAT:
ARVL_RATE:4
CPIOCYCLES:8
NO ERRORS DETECTED DURING NUMERICAL SOLUTION.

WHAT:all
```

```
ELEMENT         UTILIZATION
CPUQ            0.80000
DISKQ           0.30400


ELEMENT         THROUGHPUT
CPUQ            32.00000
DISKQ           32.00000


ELEMENT         MEAN QUEUE LENGTH
CPUQ            4.00000
DISKQ           0.66991


ELEMENT         MEAN QUEUEING TIME
CPUQ            0.12500
DISKQ           0.02093


ELEMENT         OPEN CHAIN POPULATION
TRNSACTNS       4.66991


ELEMENT         OPEN CHAIN RESPONSE TIME
TRNSACTNS       1.16748

WHAT:
ARVL_RATE:
```

A1.2. I/O Subsystem Model

The computer system model of Section 1 assumed that there was no competition between disks, e.g., for channels. Let us consider a computer system with two disks where the same channel must be used to initiate positioning (arm and/or rotational) and for transfers. If the channel is not available when a device is in the correct rotational position, a job must wait a full revolution before it can try again to get the channel and make the transfer. See Figure A1.2. This figure is similar to the Section 1 computer system model but omits the terminals queue and memory queue. This model also represents round robin scheduling at the CPU using JV(0) to maintain the remaining service time. There is a passive queue representing the channel, and there are both passive and active queues representing each device. The passive queues are used for representing contention and the active queues are used for representing timing; there will never be more than one job at the (device) active queues. After a job acquires the token for a device, it requests the channel, to initiate arm or rotational positioning. As soon as it gets the channel it releases it; we assume the time to initiate positioning is negligible, but that the time waiting to initiate positioning may not be negligible. The device arm may or may not be at the proper cylinder. We assume that with probability 1−MOVEARMP the arm is already at the right cylinder and the job only needs to wait for rotational positioning. If the arm is not at the right cylinder we assume each of the remaining cylinders is equally likely to be the correct one. Global variables are used to keep track of the current and chosen cylinder. After a seek the job initiates and waits for rotational positioning.

Whether a seek was required or not, we assume the rotational positioning time is uniformly distributed from 0 to one revolution.

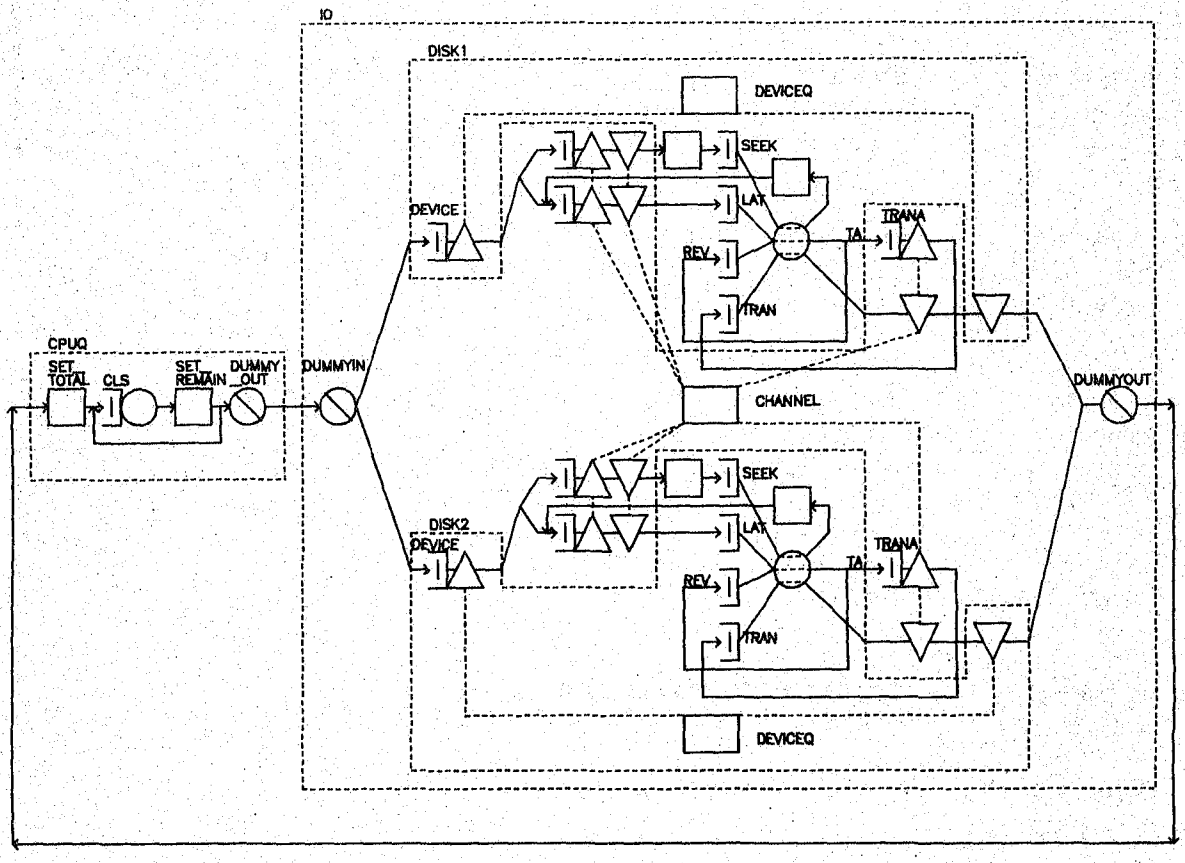

Figure A1.2 - I/O Subsystem Model

After the device is at the correct rotational position, the TA status function is used to determine whether the channel is available. If it is not, then the job is delayed for a full revolution. Once the job gets the channel, it has a transfer time (which we assume to be constant, e.g., one page) and then releases the channel and device. The degree of multiprogramming is assumed constant. The following dialogue file could be used for this model:

```
MODEL:examp2
   METHOD:simulation
   NUMERIC IDENTIFIERS:mean_serve quantum overhead
      MEAN_SERVE:.02
      QUANTUM:.02
      OVERHEAD:.0002
   SUBMODEL:rrqueue /*round robin queue*/
      NUMERIC PARAMETERS:mean_serve quantum overhead
      CHAIN PARAMETERS:chn
      QUEUE:q
         TYPE:fcfs
         CLASS LIST:cls
            SERVICE TIMES:standard(min(jv(0),quantum)+overhead,0)
      SET NODES:set_total
```

```
      ASSIGNMENT LIST:jv(0)=standard(mean_serve,1)
      SET NODES:set_remain
      ASSIGNMENT LIST:jv(0)=jv(0)-min(jv(0),quantum)
      DUMMY NODES:dummy_out
      CHAIN:chn
         TYPE:external
         INPUT:set_total
         OUTPUT:dummy_out
         :set_total->cls->set_remain->cls dummy_out;if(jv(0)>0) if(t)
   END OF SUBMODEL RRQUEUE
   SUBMODEL:iosys /*subsystem with device contention for channel*/
      CHAIN PARAMETERS:c
      NUMERIC IDENTIFIERS:movearmp
         MOVEARMP:1/3
      QUEUE:channel
         TYPE:passive
         TOKENS:1
         DSPL:fcfs
         ALLOCATE NODE LIST:pos_s_a1 pos_l_a1 trana1
            NUMBERS OF TOKENS TO ALLOCATE:1
         ALLOCATE NODE LIST:pos_s_a2 pos_l_a2 trana2
            NUMBERS OF TOKENS TO ALLOCATE:1
         RELEASE NODE LIST:pos_s_r1 pos_l_r1 tranr1
         RELEASE NODE LIST:pos_s_r2 pos_l_r2 tranr2
      DUMMY NODES:dummyin dummyout
      SUBMODEL:dasd /*individual device*/
         NUMERIC PARAMETERS:ncyl startarmt cylt revt trant
         NODE PARAMETERS:pos_s_a pos_s_r pos_l_a pos_l_r trana tranr
         CHAIN PARAMETERS:c
         GLOBAL VARIABLE IDENTIFIERS:oldcyl newcyl
            OLDCYL:ncyl/2
            NEWCYL:0
         QUEUE:deviceq
            TYPE:passive
            TOKENS:1
            DSPL:fcfs
            ALLOCATE NODE LIST:device
               NUMBERS OF TOKENS TO ALLOCATE:1
            RELEASE NODE LIST:devicer
         QUEUE:timesq
            TYPE:fcfs
            CLASS LIST:seek
               SERVICE TIMES:standard(startarmt+abs(newcyl-oldcyl) ++
                                                          *cylt,0)
            CLASS LIST:lat rev
               SERVICE TIMES:uniform(0,revt,1) standard(revt,0)
            CLASS LIST:tran
               SERVICE TIMES:standard(trant,0)
         SET NODES:setnewcyl
         ASSIGNMENT LIST:++
              newcyl=ceil(uniform(0,oldcyl-1,(oldcyl-1)/(ncyl-1);++
                          oldcyl,ncyl,(ncyl-oldcyl)/(ncyl-1)))
         SET NODES:setoldcyl
         ASSIGNMENT LIST:oldcyl=newcyl
```

```
      CHAIN:c
         TYPE:external
         INPUT:device
         OUTPUT:devicer
         :device->pos_s_a pos_l_a;movearmp 1-movearmp
         :pos_s_a->pos_s_r->setnewcyl->seek->setoldcyl->pos_l_a
         :pos_l_a->pos_l_r->lat
         :lat->trana rev;if(ta>0) if(t)
         :rev->trana rev;if(ta>0) if(t)
         :trana->tran->tranr->devicer
   END OF SUBMODEL DASD
   INVOCATION:disk1
      TYPE:dasd
      NCYL:800
      STARTARMT:.01
      CYLT:.0001
      REVT:.0166667
      TRANT:.0029
      POS_S_A:pos_s_a1
      POS_S_R:pos_s_r1
      POS_L_A:pos_l_a1
      POS_L_R:pos_l_r1
      TRANA:trana1
      TRANR:tranr1
      C:c
   INVOCATION:disk2
      TYPE:dasd
      NCYL:800
      STARTARMT:.01
      CYLT:.0001
      REVT:.0166667
      TRANT:.0029
      POS_S_A:pos_s_a2
      POS_S_R:pos_s_r2
      POS_L_A:pos_l_a2
      POS_L_R:pos_l_r2
      TRANA:trana2
      TRANR:tranr2
      C:c
   CHAIN:c
      TYPE:external
      INPUT:dummyin
      OUTPUT:dummyout
      :dummyin->disk1.input disk2.input;.5 .5
      :disk1.output disk2.output->dummyout
END OF SUBMODEL IOSYS
INVOCATION:cpuq
   TYPE:rrqueue: mean_serve; quantum; overhead; c
INVOCATION:io
   TYPE:iosys
   C:c
CHAIN:c
   TYPE:closed
   POPULATION:4
```

```
      :cpuq.output->io.input
      :io.output->cpuq.input
   CONFIDENCE INTERVAL METHOD:replications
   INITIAL STATE DEFINITION -
   CHAIN:c
      NODE LIST:cpuq.set_total
      INIT POP:4
   CONFIDENCE LEVEL:90
   NUMBER OF REPLICATIONS:5
   INITIAL PORTION DISCARDED:10 /*percent*/
   REPLIC LIMITS-
      NODES FOR DEPARTURE COUNTS:cpuq.set_total
         DEPARTURES:10000
   LIMIT - CP SECONDS:300
   TRACE:no
END
```

Since the channel is shared between the disks, the submodel representing a disk must have entry and exit points for the allocate and release nodes for the channel as well as for the allocate and release nodes for the disk. Node parameters are used for the channel allocate and release nodes.

Following is the RQ2PRNT file obtained from the EVAL command:

```
RESQ2 VERSION DATE: MARCH 3, 1982 -  TIME: 22:29:10  DATE: 03/09/82
MODEL:EXAMP2
REPLICATION   1: SET_TOTAL DEPARTURE LIMIT
REPLICATION   2: SET_TOTAL DEPARTURE LIMIT
REPLICATION   3: SET_TOTAL DEPARTURE LIMIT
REPLICATION   4: SET_TOTAL DEPARTURE LIMIT
REPLICATION   5: SET_TOTAL DEPARTURE LIMIT
NO ERRORS DETECTED DURING SIMULATION.  19837 DISCARDED EVENTS

  SIMULATED TIME PER REPLICATION:       207.52304
                        CPU TIME:          291.46
NUMBER OF EVENTS PER REPLICATION:           35782
          NUMBER OF REPLICATIONS:               5

WHAT:tpbo


INVOCATION INVOCATION ELEMENT     THROUGHPUT
           CPUQ       Q           68.54985(68.37177,68.72792) 0.5%
           IO         CHANNEL     101.30110(100.28430,102.31792) 2.0%
IO         DISK1      DEVICEQ     21.86382(21.54640,22.18124) 2.9%
IO         DISK1      TIMESQ      52.40106(51.56384,53.23827) 3.2%
IO         DISK2      DEVICEQ     21.50816(21.28053,21.73578) 2.1%
IO         DISK2      TIMESQ      51.48280(50.97803,51.98758) 2.0%
           CPUQ       SET_TOTAL   43.36867
           CPUQ       SET_REMAIN  68.54947
           CPUQ       DUMMY_OUT   43.37251
           IO         POS_S_R1    7.35436
           IO         POS_L_R1    21.86263
           IO         TRANR1      21.86166
```

```
            IO          POS_S_R2    7.20016
            IO          POS_L_R2    21.50700
            IO          TRANR2      21.50700
            IO          DUMMYIN     43.37251
            IO          DUMMYOUT    43.36867
IO          DISK1       DEVICER     21.86166
IO          DISK1       SETNEWCYL   7.35436
IO          DISK1       SETOLDCYL   7.35340
IO          DISK2       DEVICER     21.50700
IO          DISK2       SETNEWCYL   7.20016
IO          DISK2       SETOLDCYL   7.20016

WHAT:utbo(cpuq.q,io.channel,io.disk1.deviceq,io.disk2.deviceq)


INVOCATION  INVOCATION  ELEMENT     UTILIZATION
            CPUQ        Q           0.87926(0.87447,0.88406)  1.0%
            IO          CHANNEL     0.12578(0.12460,0.12696)  0.2%
IO          DISK1       DEVICEQ     0.53930(0.52858,0.55002)  2.1%
IO          DISK2       DEVICEQ     0.52685(0.52173,0.53196)  1.0%

WHAT:qlbo(*)


INVOCATION  INVOCATION  ELEMENT     MEAN QUEUE LENGTH
            CPUQ        Q           2.21399(2.18089,2.24708)  3.0%
            IO          CHANNEL     0.12935(0.12820,0.13050)  1.8%
            IO           POS_S_A1    3.65E-04(2.95E-04,4.34E-04)  38.0%
            IO           POS_L_A1    1.39E-03(1.30E-03,1.48E-03)  13.1%
            IO           TRANA1      0.06341(0.06248,0.06433)  2.9%
            IO           POS_S_A2    4.02E-04(3.75E-04,4.29E-04)  13.3%
            IO           POS_L_A2    1.42E-03(1.32E-03,1.51E-03)  13.3%
            IO           TRANA2      0.06237(0.06171,0.06303)  2.1%
IO          DISK1       DEVICEQ     0.91284(0.87976,0.94592)  7.2%
IO          DISK1       TIMESQ      0.53754(0.52681,0.54828)  4.0%
IO          DISK1        SEEK        0.27007(0.26338,0.27675)  5.0%
IO          DISK1        LAT         0.18209(0.17829,0.18589)  4.2%
IO          DISK1        REV         0.02198(0.02002,0.02394)  17.8%
IO          DISK1        TRAN        0.06341(0.06248,0.06433)  2.9%
IO          DISK2       DEVICEQ     0.87317(0.86327,0.88307)  2.3%
IO          DISK2       TIMESQ      0.52503(0.51989,0.53017)  2.0%
IO          DISK2        SEEK        0.26165(0.25638,0.26693)  4.0%
IO          DISK2        LAT         0.17991(0.17757,0.18225)  2.6%
IO          DISK2        REV         0.02109(0.02007,0.02212)  9.7%
IO          DISK2        TRAN        0.06237(0.06171,0.06303)  2.1%

WHAT:st(*)


INVOCATION  INVOCATION  ELEMENT     MEAN SERVICE TIMES
            CPUQ        Q           0.01283
IO          DISK1       TIMESQ      0.01026
IO          DISK1        SEEK        0.03671
IO          DISK1        LAT         8.3271E-03
```

```
IO          DISK1        REV          0.01667
IO          DISK1        TRAN         2.9000E-03
IO          DISK2        TIMESQ       0.01020
IO          DISK2        SEEK         0.03634
IO          DISK2        LAT          8.3646E-03
IO          DISK2        REV          0.01667
IO          DISK2        TRAN         2.9000E-03

WHAT:gv

INVOCATION  INVOCATION   ELEMENT      FINAL VALUES OF GLOBAL VARIABLES
IO          DISK1        OLDCYL       119.00000
IO          DISK1        NEWCYL       119.00000
IO          DISK2        OLDCYL       645.00000
IO          DISK2        NEWCYL       645.00000

WHAT:
```

Some blank columns and less significant digits have been edited from this copy of the file to allow presentation within the column width used in this document.

A1.3. Communication Protocol Model

Like the example of Section 1, the example of this section considers terminals connected to an interactive computing system. However, the model of Section 1 emphasized representation of the computer system and ignored communication between the terminals and the computer system. The model of this section will ignore details of the computer system, representing the computer system by a single queue with queue length dependent service rates, and will focus on communication between the terminals and computing system.

We assume the terminals are organized in three separate groups. (The submodel definitions given below would apply with any number of groups. Minor modifications in the main model would be needed to change the number of groups.) The terminals share a full duplex 2400 baud line to the computer system. In order to avoid conflicts between traffic destined from a terminal group to the computer system, a polling protocol gives each group a turn to transmit any traffic it has for the computing system. The messages sent from the terminals to the computing system are fairly short with a maximum length of 640 bits. However, the messages sent from the computing system to terminals are longer and more variable in length, with a mean length of 800 bits. To prevent a long message from monopolizing the line from the computing system to the terminals, the messages are divided into packets of maximum length 256 bits. Only 240 of the 256 bits are used for data, with the remaining bits used for control information. To prevent a terminal controller from receiving more data than it can handle, a simple window flow control protocol is used. The protocol allows only a single message (typically, several packets) to be sent to a terminal group before that group explicitly requests another message be sent.

The model consists of three submodels, a queue representing the computer system and a passive queue used for measuring response times. The first submodel, TERM_GROUP, represents a terminal group. There will be one invocation of TERM_GROUP for each group. The second submodel, POLL_LINE, represents the communication line. There is just one inovcation of POLL_LINE. The third submodel, FLOW_N_PKT, represents the

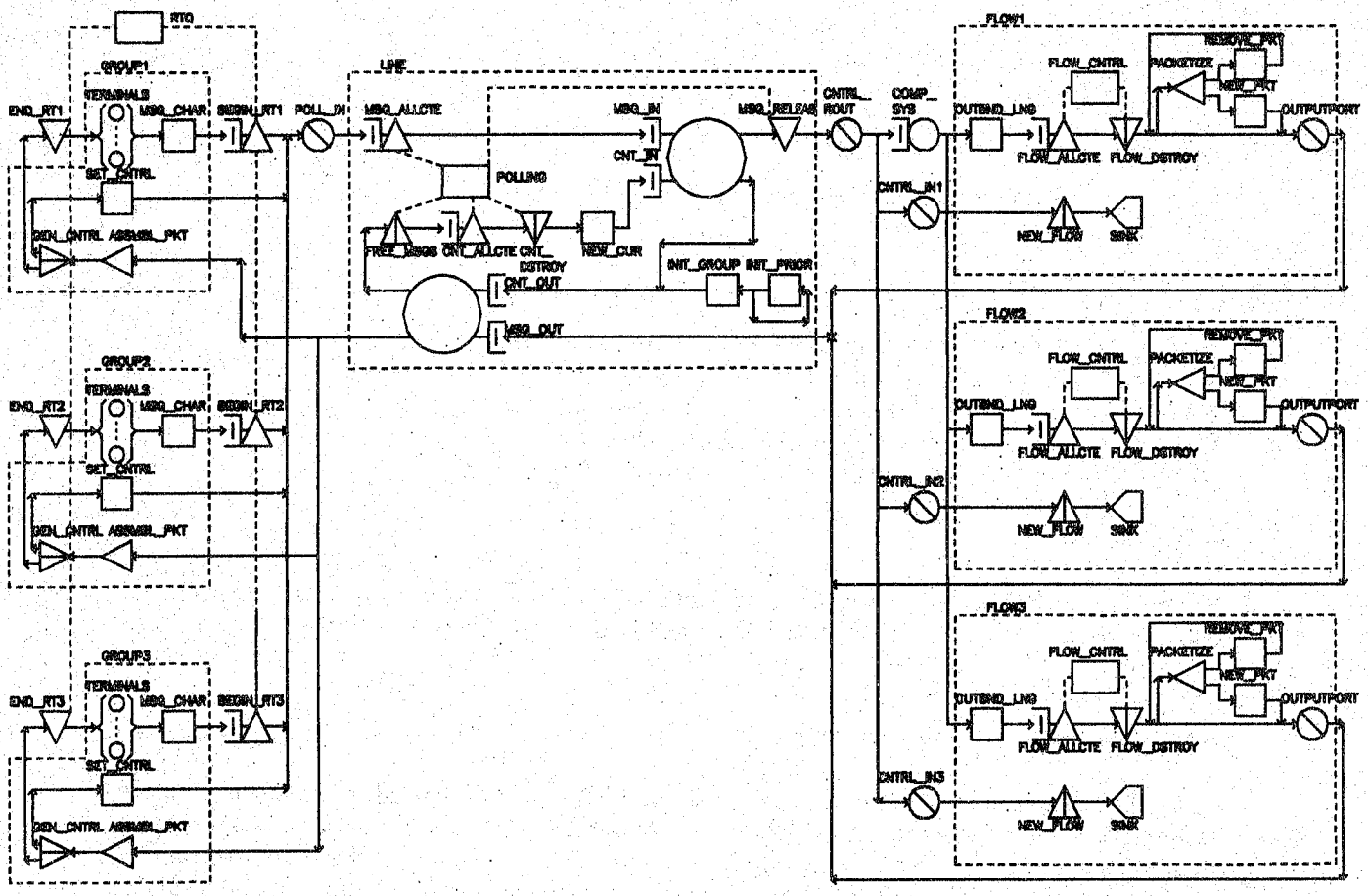

Figure A1.3 - Communication Protocol Model

window flow control protocol and the division of messages into packets. There will be one invocation of FLOW__N__PKT for each terminal group.

Following is a dialogue file for definition of TERM__GROUP. (Some of the names in this dialogue file are names of numeric identifiers declared in the invoking model.)

```
SUBMODEL:term_group
   NUMERIC PARAMETERS:group_no
   NODE PARAMETERS:begin_rt end_rt
   CHAIN PARAMETERS:c
   QUEUE:terminalsq
      TYPE:is
      CLASS LIST:terminals
         SERVICE TIMES:thinktime
   SET NODES:msg_char /*message characteristics*/
   ASSIGNMENT LIST:jv(group)=group_no                    ++
                   jv(msg_type)=data                     ++
                   jv(msg_leng)=uniform(24,640,1)
   SET NODES:set_cntrl
   ASSIGNMENT LIST:jv(group)=group_no                    ++
                   jv(msg_type)=control                  ++
                   jv(msg_leng)=32
```

```
    SPLIT NODES:gen_cntrl
    FUSION NODES:assmbl_pkt
    CHAIN:c
       TYPE:external
       INPUT:assmbl_pkt
       OUTPUT:set_cntrl
       :assmbl_pkt->gen_cntrl->end_rt set_cntrl;split
       :end_rt->terminals->msg_char->begin_rt
END OF SUBMODEL TERM_GROUP
```

Jobs are initially placed at the terminals to represent users. At the end of a think time (and keying time), a job goes to set node MSG__CHAR which sets job variables giving the message characteristics, i.e., the group producing the message, the fact that this is a data message (as opposed to a control message for the window flow control protocol), and the message length. The job then goes to node parameter BEGIN__RT which is an allocate node for response time measurement. Jobs representing packets returning from the computing system go to fusion node ASSMBL__PKT. When all packets of a message have arrived at the fusion node, a single job representing the assembled message leaves the fusion node. That job goes to split node GEN__CNTRL to generate a control message which will eventually allow another message to be sent, as discussed below. The control message job goes to set node SET__CNTRL which sets the job variables giving its characteristics. From the set node the control message job will go to the communication line. The job representing the message goes to node parameter END__RT, a release node for response time measurement, and then goes to the terminals.

Following is a dialogue file for definition of POLL__LINE. (Some of the names in this dialogue file are names of numeric identifiers declared in the invoking model.)

```
SUBMODEL:poll_line
   NUMERIC PARAMETERS:no_groups
   NODE PARAMETERS:inboundin inboundout
   CHAIN PARAMETERS:c
   GLOBAL VARIABLES:cur_group cur_prior(no_groups)
      CUR_GROUP:1
      CUR_PRIOR:0
   QUEUE:polling
      TYPE:passive
      TOKENS:0
      DSPL:prty
      ALLOCATE NODE LIST:msg_allcte
         NUMBERS OF TOKENS TO ALLOCATE:1
         PRIORITIES:cur_prior(jv(group))+jv(msg_type)
      ALLOCATE NODE LIST:cnt_allcte
         NUMBERS OF TOKENS TO ALLOCATE:1
         PRIORITIES:cur_prior(cur_group)+2
      RELEASE NODE LIST:msg_releas
      DESTROY NODE LIST:cnt_dstroy
      CREATE NODE LIST:free_msgs
         NUMBERS OF TOKENS TO CREATE:1
   QUEUE:inbound
      TYPE:fcfs
      CLASS LIST:msg_in
         SERVICE TIMES:standard(jv(msg_leng),0)/2400
      CLASS LIST:cnt_in
```

```
        SERVICE TIMES:32/2400
    QUEUE:outbound
       TYPE:prty
       CLASS LIST:msg_out
          SERVICE TIMES:standard(jv(msg_leng),0)/2400
          PRIORITIES:2
       CLASS LIST:cnt_out
          SERVICE TIMES:32/2400
          PRIORITIES:1
    SET NODES:new_cur
    ASSIGNMENT LIST:cur_prior(cur_group)=                              ++
                              cur_prior(cur_group)+3*no_groups         ++
                    cur_group=(cur_group mod no_groups)+1
    SET NODES:init_prior
    ASSIGNMENT LIST:cur_prior(cur_group)=cur_group*3-2                 ++
                    cur_group=cur_group+1
    SET NODES:init_group
    ASSIGNMENT LIST:cur_group=1
    CHAIN:c
       TYPE:external
       INPUT:msg_out
       OUTPUT:msg_out
       :inboundin->msg_allcte->msg_in->msg_releas->inboundout
    CHAIN:pollingjob
       TYPE:closed
       POPULATION:1
       :init_prior->init_prior;if(cur_group<=no_groups)
       :init_prior->init_group;if(t)
       :init_group->cnt_out->free_msgs->cnt_allcte->cnt_dstroy
       :cnt_dstroy->new_cur->cnt_in->cnt_out
END OF SUBMODEL POLL_LINE
```

The key element of this submodel is the use of the vector of priorities, CUR__PRIOR, which is used with passive queue POLLING. There are three priority levels for a group, high priority for the flow control messages, medium priority for data messages and low priority for the polling job. Group i has highest priority given by CUR__PRIOR(i) for flow control messages, priority CUR__PRIOR(i)+1 for data messages and priority CUR__PRIOR(i)+2 for the polling job. Polling is accomplished by the polling job creating a token at node FREE__MSGS and then waiting at allocate node CNT__ALLCTE until all higher priority jobs (flow control and data messages for the group being polled) have received the token, spent a service time at class MSG__IN and then released the token at MSG__RELEAS. When the polling job receives the token, it increases the CUR__PRIOR value for the group just polled by 3×NO__GROUPS, thus giving the group just polled the lowest base priority. (Since the priority value may be any integer up to $2^{31}-1$, the values in CUR__PRIOR can be increased indefinitely without fear of overflow in a feasible run length. However, the values in CUR__PRIOR could be reset to their initial values periodically if overflow was a concern.) The relatively imitative representation of polling used in the definition of POLL__LINE is expensive in terms of simulated events, because of the polling that occurs when there are no waiting jobs. Alternate, but more complex, representations may reduce the simulation expense.

Following is a dialogue file for definition of FLOW__N__PKT. (Some of the names in this dialogue file are names of numeric identifiers declared in the invoking model.)

```
SUBMODEL:flow_n_pkt
   NODE PARAMETERS:cntrl_in
   CHAIN PARAMETERS:c
   QUEUE:flow_cntrl
      TYPE:passive
      TOKENS:1
      DSPL:fcfs
      ALLOCATE NODE LIST:flowallcte
         NUMBERS OF TOKENS TO ALLOCATE:1
      DESTROY NODE LIST:flowdstroy
      CREATE NODE LIST:new_flow
         NUMBERS OF TOKENS TO CREATE:1
   SET NODES:outbnd_lng
   ASSIGNMENT LIST:jv(msg_leng)=standard(800,1)
   SET NODES:remove_pkt
   ASSIGNMENT LIST:jv(msg_leng)=jv(msg_leng)-240
   SET NODES:new_pkt
   ASSIGNMENT LIST:jv(msg_leng)=256
   FISSION NODES:packetize
   DUMMY NODES:outputport
   CHAIN:c
      TYPE:external
      INPUT:outbnd_lng
      OUTPUT:outputport
      :outbnd_lng->flowallcte->flowdstroy
      :flowdstroy->packetize outputport;if(jv(msg_leng)>256) if(t)
      :packetize->remove_pkt new_pkt;fission
      :remove_pkt->packetize outputport;if(jv(msg_leng)>256) if(t)
      :new_pkt->outputport
      :cntrl_in->new_flow->sink
END OF SUBMODEL FLOW_N_PKT
```

A job representing a message from the computer system goes to set node OUTBND__LNG to establish the length of the message. The job then goes to allocate node FLOWALLCTE to wait for a token. A token will be made available by a job representing a (flow control message) arriving from node parameter CNTRL__IN and going to create node NEW__FLOW. (That job then goes to the sink.) When a job waiting at FLOWALLCTE gets a token, it will then generate new jobs represting packets (if necessary) at fission node PACKETIZE. Set node REMOVE__PKT decrements the message length by 240 (the number of data bits in a packet) and set node NEW__PKT sets the new packet's JV(MSG__LNG) to 256 (data bits plus control bits).

Following is a dialogue file for definition of the main model.

```
MODEL:EXAMP3
   /* Computer system with several remote terminal groups. */
   /* Groups connected to system by polled communication   */
   /* line.  Flow control and packetizing of messages.     */
   METHOD:simulation
   NUMERIC IDENTIFIERS:no_terms /*per group*/ thinktime
      NO_TERMS:10
      THINKTIME:20
   NUMERIC IDENTIFIERS:control data
      CONTROL:0   /*Code to be used for control messages*/
```

```
   DATA:1       /*Code to be used for data messages*/
NUMERIC IDENTIFIERS:group msg_type msg_leng
   GROUP:0      /*JV to be used to indicate group*/
   MSG_TYPE:1   /*JV to be used to indicate type*/
   MSG_LENG:2   /*JV to be used to indicate length*/
MAX JV:2
QUEUE:rtq /*response time*/
   TYPE:passive
   TOKENS:2147483647 /*"infinity"*/
   DSPL:fcfs
   ALLOCATE NODE LIST:begin_rt1 begin_rt2 begin_rt3
      NUMBERS OF TOKENS TO ALLOCATE:1
   RELEASE NODE LIST: end_rt1   end_rt2   end_rt3
QUEUE:comp_sysq
   TYPE:active
   DSPL:ps
   CLASS LIST:comp_sys
      WORK DEMANDS:1
   SERVER-
      RATE:1.4 2.0 2.25 2.4
DUMMY NODES:poll_in cntrl_rout cntrl_in1 cntrl_in2 cntrl_in3
INCLUDE:termgrp
INCLUDE:pollline
INCLUDE:flownpkt
INVOCATION:group1
   TYPE:term_group
   GROUP_NO:1
   BEGIN_RT:begin_rt1
   END_RT:end_rt1
   C:c
INVOCATION:group2
   TYPE:term_group
   GROUP_NO:2
   BEGIN_RT:begin_rt2
   END_RT:end_rt2
   C:c
INVOCATION:group3
   TYPE:term_group
   GROUP_NO:3
   BEGIN_RT:begin_rt3
   END_RT:end_rt3
   C:c
INVOCATION:line
   TYPE:poll_line
   NO_GROUPS:3
   INBOUNDIN:poll_in
   INBOUNDOUT:cntrl_rout
   C:c
INVOCATION:flow1
   TYPE:flow_n_pkt
   CNTRL_IN:cntrl_in1
   C:c
INVOCATION:flow2
   TYPE:flow_n_pkt
```

```
      CNTRL_IN:cntrl_in2
      C:c
   INVOCATION:flow3
      TYPE:flow_n_pkt
      CNTRL_IN:cntrl_in3
      C:c
   CHAIN:c
      TYPE:open
      :begin_rt1 begin_rt2 begin_rt3->poll_in
      :cntrl_rout->comp_sys;if(jv(msg_type)=data)
      :cntrl_rout->cntrl_in1;if(jv(group)=1)
      :cntrl_rout->cntrl_in2;if(jv(group)=2)
      :cntrl_rout->cntrl_in3;if(jv(group)=3)
      :comp_sys->flow1.input;if(jv(group)=1)
      :comp_sys->flow2.input;if(jv(group)=2)
      :comp_sys->flow3.input;if(jv(group)=3)
      :flow1.output flow2.output flow3.output->line.input
      :line.output->group1.input;if(jv(group)=1)
      :line.output->group2.input;if(jv(group)=2)
      :line.output->group3.input;if(jv(group)=3)
      :group1.output group2.output group3.output->poll_in
   QUEUES FOR QUEUEING TIME DIST:rtq
      VALUES:.5 1 2 4 8
   NODES FOR QUEUEING TIME DIST:begin_rt1 begin_rt2 begin_rt3
      VALUES:.5 1 2 4 8
   CONFIDENCE INTERVAL METHOD:spectral
   INITIAL STATE DEFINITION-
   CHAIN:line.pollingjob
      NODE LIST:line.init_prior
      INIT POP:1
   CHAIN:c
      NODE LIST:group1.terminals group2.terminals group3.terminals
      INIT POP: no_terms         no_terms         no_terms
   CONFIDENCE LEVEL:90
   SEQUENTIAL STOPPING RULE:yes
      CONFIDENCE INTERVAL QUEUES:rtq rtq comp_sysq
         MEASURES:               qt  qtd qt
         ALLOWED WIDTHS:         10  10  10
      CONFIDENCE INTERVAL NODES:begin_rt1 begin_rt2 begin_rt3
         MEASURES:              qt        qt        qt
         ALLOWED WIDTHS:        100       100       100
   INITIAL PORTION DISCARDED:10
   INITIAL PERIOD LIMITS-
      QUEUES FOR DEPARTURE COUNTS:rtq
         DEPARTURES:1000
   LIMIT - CP SECONDS:500
   TRACE:no
END
```

This dialogue defines the queues for response time measurement and the computer system, invokes the submodels and defines the connections between the invocations.

Following is an RQ2PRNT file for this model as produced by EVAL.

```
RESQ2 VERSION DATE: APRIL 3, 1982 -  TIME: 17:56:53  DATE: 04/03/82
MODEL:EXAMP3
SAMPLING PERIOD END: RTQ DEPARTURE LIMIT
SAMPLING PERIOD END: RTQ DEPARTURE LIMIT
SAMPLING PERIOD END: RTQ DEPARTURE LIMIT
SAMPLING PERIOD END: RTQ DEPARTURE LIMIT
SAMPLING PERIOD END: RTQ DEPARTURE LIMIT
SAMPLING PERIOD END: RTQ DEPARTURE LIMIT
NO ERRORS DETECTED DURING SIMULATION.   2930 DISCARDED EVENTS

                  SIMULATED TIME:     5028.19141
                        CPU TIME:          381.07
                 NUMBER OF EVENTS:          227673

WHAT:ut(line.msg_in,line.cnt_in,line.msg_out,line.cnt_out)


INVOCATION      ELEMENT         UTILIZATION
LINE             MSG_IN          0.20631
LINE             CNT_IN          0.23099
LINE             MSG_OUT         0.48462
LINE             CNT_OUT         0.23007

WHAT:tp(rtq,begin_rt1,begin_rt2,begin_rt3)


INVOCATION      ELEMENT         THROUGHPUT
                RTQ             1.35874
                 BEGIN_RT1       0.46657
                 BEGIN_RT2       0.44887
                 BEGIN_RT3       0.44330

WHAT:qtbo(rtq,begin_rt1,begin_rt2,begin_rt3,comp_sysq)


INVOCATION      ELEMENT         MEAN QUEUEING TIME
                RTQ             2.30391(2.21360,2.39422) 7.8%
                 BEGIN_RT1       2.29731(2.21029,2.38434) 7.6%
                 BEGIN_RT2       2.35772(2.27390,2.44153) 7.1%
                 BEGIN_RT3       2.25636(2.15468,2.35804) 9.0%
                COMP_SYSQ       1.20234(1.14318,1.26150) 9.8%

WHAT:ql(rtq,begin_rt1,begin_rt2,begin_rt3,comp_sysq)


INVOCATION      ELEMENT         MEAN QUEUE LENGTH
                RTQ             3.13178
                 BEGIN_RT1       1.07279
                 BEGIN_RT2       1.05869
                 BEGIN_RT3       1.00030
                COMP_SYSQ       1.63460

WHAT:
CONTINUE RUN:yes
```

```
EXTRA SAMPLING PERIODS:1
LIMIT - CP SECONDS:1000

SAMPLING PERIOD END: RTQ DEPARTURE LIMIT
SAMPLING PERIOD END: RTQ DEPARTURE LIMIT
SAMPLING PERIOD END: RTQ DEPARTURE LIMIT
SAMPLING PERIOD END: RTQ DEPARTURE LIMIT
SAMPLING PERIOD END: RTQ DEPARTURE LIMIT
SAMPLING PERIOD END: RTQ DEPARTURE LIMIT
SAMPLING PERIOD END: RTQ DEPARTURE LIMIT
NO ERRORS DETECTED DURING SIMULATION.   2930 DISCARDED EVENTS

                SIMULATED TIME:     7542.98047
                      CPU TIME:       570.16
               NUMBER OF EVENTS:       343411

WHAT:ut(line.msg_in,line.cnt_in,line.msg_out,line.cnt_out)


INVOCATION     ELEMENT        UTILIZATION
LINE            MSG_IN         0.20599
LINE            CNT_IN         0.23301
LINE            MSG_OUT        0.47914
LINE            CNT_OUT        0.23180

WHAT:tp(rtq,begin_rt1,begin_rt2,begin_rt3)


INVOCATION     ELEMENT        THROUGHPUT
               RTQ            1.35861
                BEGIN_RT1      0.45274
                BEGIN_RT2      0.45141
                BEGIN_RT3      0.45446

WHAT:qtbo(rtq,begin_rt1,begin_rt2,begin_rt3,comp_sysq)


INVOCATION     ELEMENT        MEAN QUEUEING TIME
               RTQ            2.27488(2.21307,2.33669) 5.4%
                BEGIN_RT1      2.27176(2.20855,2.33498) 5.6%
                BEGIN_RT2      2.31046(2.22283,2.39809) 7.6%
                BEGIN_RT3      2.24264(2.14938,2.33590) 8.3%
               COMP_SYSQ      1.19929(1.16632,1.23225) 5.5%

WHAT:ql(rtq,begin_rt1,begin_rt2,begin_rt3,comp_sysq)


INVOCATION     ELEMENT        MEAN QUEUE LENGTH
               RTQ            3.09146
                BEGIN_RT1      1.02852
                BEGIN_RT2      1.04340
                BEGIN_RT3      1.01954
               COMP_SYSQ      1.62968
```

```
WHAT:qtdbo(*)

INVOCATION      ELEMENT         QUEUEING TIME DISTRIBUTION
                RTQ             5.00E-01:0.03708(0.03404,0.04012) 0.6%
                                1.00E+00:0.19633(0.18753,0.20513) 1.8%
                                2.00E+00:0.54167(0.52525,0.55808) 3.3%
                                4.00E+00:0.87529(0.86545,0.88513) 2.0%
                                8.00E+00:0.98985(0.98771,0.99199) 0.4%
                 BEGIN_RT1       5.00E-01:0.03748(0.02998,0.04498) 1.5%
                                 1.00E+00:0.19590(0.18600,0.20580) 2.0%
                                 2.00E+00:0.53206(0.51399,0.55013) 3.6%
                                 4.00E+00:0.87877(0.86588,0.89166) 2.6%
                                 8.00E+00:0.99180(0.98863,0.99497) 0.6%
                 BEGIN_RT2       5.00E-01:0.03465(0.03048,0.03883) 0.8%
                                 1.00E+00:0.18767(0.16722,0.20811) 4.1%
                                 2.00E+00:0.52658(0.49773,0.55543) 5.8%
                                 4.00E+00:0.86990(0.85482,0.88498) 3.0%
                                 8.00E+00:0.98913(0.98526,0.99301) 0.8%
                 BEGIN_RT3       5.00E-01:0.03909(0.03283,0.04535) 1.3%
                                 1.00E+00:0.20537(0.18920,0.22153) 3.2%
                                 2.00E+00:0.56622(0.54158,0.59086) 4.9%
                                 4.00E+00:0.87719(0.86155,0.89283) 3.1%
                                 8.00E+00:0.98862(0.98558,0.99166) 0.6%

WHAT:qt(line.msg_allcte,line.msg_out)

INVOCATION      ELEMENT         MEAN QUEUEING TIME
LINE             MSG_ALLCTE      0.19342
LINE             MSG_OUT         0.58095

WHAT:ql(flow1.flowallcte,flow2.flowallcte,flow3.flowallcte)

INVOCATION      ELEMENT         MEAN QUEUE LENGTH
FLOW1            FLOWALLCTE      0.10062
FLOW2            FLOWALLCTE      0.10598
FLOW3            FLOWALLCTE      0.10036

WHAT:gv

INVOCATION      ELEMENT         FINAL VALUES OF GLOBAL VARIABLES
LINE            CUR_GROUP       2.00000
LINE            CUR_PRIOR(1)    3.9830E+05
LINE            CUR_PRIOR(2)    3.9829E+05
LINE            CUR_PRIOR(3)    3.9829E+05

WHAT:
CONTINUE RUN:no
```

APPENDIX 2 - NAMES AND KEYWORDS

A RESQ name may be any string beginning with a letter and consisting entirely of letters, digits and the characters "$" and "_" with the following restrictions:

1. Names are restricted to at most ten characters. The translator will accept longer names, but will print a warning message and ignore the extra characters.

2. Names used for model names and library members must be restricted to at most eight characters.

3. The name used for the model name and names used for submodel and queue type names may not be reused. Other names may be reused in submodels according to traditional rules of block structured programming languages such as PL/I, i.e., a name may be reused within a submodel even though it exists with entirely different meaning outside of the submodel.

4. The following keywords may not be used as names:

ABS	DESTROY	INTERVAL	OR
ACCEPTS	DISCARDED	INVOCATION	OUTPUT
ACTIVE	DISCRETE	IS	PARAMETER
ALL	DIST	ISQDEP	PARAMETERS
ALLOCATE	DISTRIBUTION	JFCKTRACE	PASSIVE
ALLOWED	DSPL	JOB	PERIOD
AMOUNTS	DUMMY	JV	PERIODS
AND	EDIT	LCFS	POP
APLOMB	END	LENGTH	POPULATION
APPROXIMATE	EVENT	LEVEL	PREEMPT
ARRAYS	EVENTS	LIMIT	PRINT
ARRIVAL	EXP	LIMITS	PRIORITIES
BE	EXTERNAL	LIST	PRTY
BY	EXTRA	LN	PRTYPR
CEIL	F	LQ	PS
CHAIN	FCFS	LRTF	QL
CHAINS	FF	MAX	QLD
CHANGE	FISSION	MEASURES	QNET4
CHECKED	FLOOR	METHOD	QT
CLASS	FOR	MIN	QTD
CLOSED	FRACTION	MODEL	QUEUE
COMPQF	FUSION	MOVEMENT	QUEUEING
COMPQP	GAMMA	MVA	QUEUES
CONFIDENCE	GLOBAL	NO	QUIT
CONVOLUTION	GUIDELINES	NODE	RATES
COUNTS	HANDLING	NODES	REGEN
CP	HOW	NONE	REGENERATION
CREATE	IDENTIFIER	NOT	RELEASE
CV	IF	NUMBER	REPLIC
CYCLES	INCLUDE	NUMERIC	REPLICATION
DEFINITION	INIT	NUMERICAL	REVIEW
DELAYS	INITIAL	OF	RJ
DEMANDS	INITIALLY	OFF	RJQ
DEPARTURE	INPUT	ON	RULE
DEPARTURES	INTERNAL	OPEN	RUN

SA	SPECTRAL	TIMES	TURN
SAMPLING	SPLIT	TO	TYPE
SAVE	SQ	TOKEN	UNIFORM
SCALED	SRTF	TOKENS	USE
SECONDS	STANDARD	TOTAL	USER
SEED	STATE	TP	UT
SEQUENTIAL	STOPPING	TQ	VALUE
SERVER	STRING	TRACE	VALUES
SERVERS	SUBMODEL	TRANSFER	VARIABLE
SET	SUBSTITUTION	TREE	VARIABLES
SIMULATED	T	TT	WIDTHS
SIMULATION	TA	TTD	WORK
SINK	TEMPLATE	TU	YES
SNAPSHOTS	TH	TUD	
SOURCE	TIME		

Where the plural form of a keyword is listed but the singular is not, the singular form may be used instead of the plural, e.g., DEMAND may be used instead of DEMANDS, but PARAMETER may not be used instead of PARAMETERS. In such cases the singular form may not be used as an name, even though it is not explicitly listed above.

5. The following global variable names have special meaning. They should not be used as global variable names unless the special meaning is intended.

CLOCK - This global variable contains current simulated time. CLOCK must be initialized to zero (0). CLOCK is available only for reference within expressions and should not be used as the variable to be assigned by a set node. *Any attempt to assign a value to CLOCK during simulation will abort the run.*

TRACEON - If TRACEON is set to a posive value, by initialization or by a set node, simulation trace output will be produced. TRACEON overrides the "INITIALLY ON:" reply. TRACEON is set to 1 at when trace is turned on by the "TURN TRACE ON-" specification and is set to 0 when trace is turned off by the "TURN TRACE OFF-" specification.

JOBTRACE - If JOBTRACE is set to a positive value, job movement trace will be produced, provided that trace has been turned on by the "TURN TRACE ON-" specification or by assignment or initialization of TRACEON. JOBTRACE overrides the "JOB MOVEMENT:" specification.

QUEUETRACE - If QUEUETRACE is set to a positive value, queue trace will be produced, for all queues, provided that trace has been turned on by the "TURN TRACE ON-" specification or by assignment or initialization of TRACEON. QUEUETRACE overrides the "QUEUES:" specification.

EVENTTRACE - If EVENTTRACE is set to a positive value, event handling trace will be produced, provided that trace has been turned on by the "TURN TRACE ON-" specification or by assignment

or initialization of TRACEON. EVENTTRACE overrides the "EVENT HANDLING:" specification.

LISTTRACE - If LISTTRACE is set to a positive value, event list trace will be produced, provided that trace has been turned on by the "TURN TRACE ON-" specification or by assignment or initialization of TRACEON. LISTTRACE overrides the "EVENT LIST:" specification.

SNAPTRACE - If SNAPTRACE is set to a positive value, snapshot trace will be produced, provided that trace has been turned on by the "TURN TRACE ON-" specification or by assignment or initialization of TRACEON. SNAPTRACE overrides the "SNAPSHOTS:" specification.

EXPERTRACE - This global variable is reserved for use by RESQ developers.

SAUERTRACE - This global variable is reserved for use by RESQ developers.

APPENDIX 3 - EXPRESSIONS

RESQ expressions correspond to those of programming languages, with essentially the same rules as languages such as PL/I, Pascal and Fortran (but not APL). Section A3.5 discusses RESQ expressions largely from the point of view of expression execution. It is intended to be informal; a more formal definition of RESQ expressions is given in the grammar in Appendix 4. Except for expressions used in routing predicates, any expression in RESQ must be such that it can be evaluated to a scalar numeric value, a vector of numeric values or a matrix of numeric values. Section A3.6 discusses expressions for routing predicates.

"Simulation dependent" expressions are those that depend on job variables, chain variables, global variables, distribution keywords (Section A3.1), the USER function (Section A3.2), status functions (Section A3.3) or the PRINT function (Section A3.4). Except where otherwise noted, simulation dependent expressions may be used anywhere in the definition of a simulation model. Expressions which are not simulation dependent are "simulation independent." Only simulation independent expressions may be used in numerically solved models.

A3.1. Distribution Functions

When one has little information about random values other than mean values, then it is reasonable to arbitrarily assume that the random values have a distribution which is completely specified by the mean, e.g., the (negative) exponential distribution. However, when one has more information, then one would usually like to have a representation which includes that information. For example, if one knows standard deviations, then one would like to include standard deviations in a model. RESQ provides a standardized distribution form which is completely specified by the mean and coefficient of variation and which is expedient for simulation and confidence interval estimation. (The coefficient of variation is defined as the standard deviation divided by the mean.) The RESQ STANDARD distribution will often be sufficient. However, if the user has additional information then the user may wish to try to fit the distribution more precisely. The DISCRETE distribution provides one mechanism for doing this, i.e., the user supplies RESQ with a table of values and associated probabilities. If the discrete distribution is not appropriate or convenient, then one of the more detailed continuous forms provided by RESQ, the BE (Branching Erlang) or the UNIFORM, may be appropriate. Other distributions are indirectly available, and we will give examples of how indirect definition of distributions may be accomplished. If none of these options are satisfactory for a particular model, the user has the option of defining a PL/I procedure to provide distribution values. This can be done with the USER function described in Section A3.2.

We next describe the full generality of the BE and UNIFORM distributions provided by RESQ. We then discuss the RESQ STANDARD distribution and how the BE and UNIFORM distributions are used in defining this form. We then discuss the DISCRETE distribution in more detail and discuss the indirect definition of distributions.

A3.1.1. BE (Branching Erlang) Distribution

A number of distribution forms can be grouped together as representatives of the *method of exponential stages.* Perhaps the best known of these are the Erlang distribution, the hypoexponential distribution and the hyperexponential distribution. The branching Erlang distribution is less well known but includes all three of the above distributions and many other distributions as special cases. (The branching Erlang form was originally proposed by Cox. He showed that by using the artifice of complex "probabilities" and holding "times" that the branching Erlang form can be used to represent arbitrary distributions with rational Laplace

transforms. Of course, one cannot simulate complex "probabilities" or holding "times." The branching Erlang form is quite general without the use of complex values.) Figure A3.1 illustrates the branching Erlang form.

The BE distribution may be thought of as consisting of K exponential stages (which are represented by circles in the figure). Stage i, $i = 1,...,K$, has a mean (exponential time) m_i and a "branching" probability (to be described shortly) p_i. A sample from the distribution consists of the sum of (independent) samples from stages 1 to k where k is between 1 and K and selected by the following rule: With probability p_1, k is chosen to be 1, with probability $(1 - p_1)p_2$, k is chosen to be 2, ... and with probability $(1 - p_1)(1 - p_2)...(1 - p_{K-1})$, k is chosen to be K. In other words, p_i is the probability of branching past the stages after stage i. Note that p_K is identically 1. The mean, M, of the BE distribution is given by

$$M = \sum_{k=1}^{K} (1 - p_1)(1 - p_2)...(1 - p_{k-1})p_k \sum_{i=1}^{k} m_i \qquad (A3.1)$$

and the coefficient of variation, C, is given by

$$C = \frac{\sqrt{\sum_{k=1}^{K} (1-p_1)(1-p_2)...(1-p_{k-1})p_k \left[\sum_{i=1}^{k} m_i^2 + \left(\sum_{i=1}^{k} m_i \right)^2 \right] - M^2}}{M}. \qquad (A3.2)$$

The BE distribution reduces to the exponential distribution if we set p_1 to 1 and m_1 to M where M is the mean of the distribution. The BE distribution reduces to the Erlang distribution if we set p_i to zero for all i other than K and set m_i to M/K. The hypoexponential distribution is a generalization of the Erlang distribution which does not require equality of the stage means $\{m_i\}$. A 2 stage hyperexponential distribution can be thought of as a choice of an exponential distribution with mean m_1 with probability q and a choice of an exponential distribution with mean m_2 otherwise. Without loss of generality we may assume $m_1 < m_2$. Then the BE distribution with 2 stages and the corresponding stage means is equivalent to the hyperexponential if we set p_1 to $q + (1 - q)m_1/m_2$. (Note that if we wish to have the classical representation of hyperexponential service times at a queue we can accomplish this by having two classes with exponential distributions with means m_1 and m_2 and routing a job to the first class with probability q and to the second class otherwise.)

In RESQ the BE distribution is represented by the keyword "bE" followed by a parenthesized list of pairs of stage means and probabilities where the pairs are separated by semi-colons (";"), i.e.,

$$\text{bE}(m_1,p_1; \ldots ; m_K,p_K).$$

(We make "E" upper case in deference to Erlang, but the program does not require this. As we said before, there is no internal distinction between upper and lower case.) As is true throughout RESQ, commas (",") may be replaced by blanks.

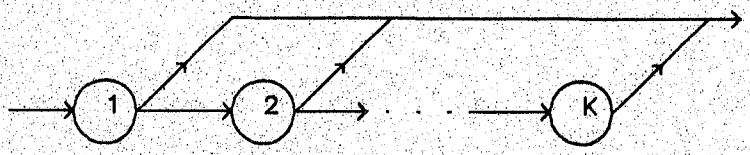

Figure A3.1 - BE (Branching Erlang) Distribution

The remainder of Section A3.1.1 applies only to simulations using the regenerative method for confidence intervals. With the regenerative method with BE arrival and/or service times, it is necessary to consider the distribution stages in determining regeneration states, but this is hidden from the user. The simulation samples the arrival distribution a stage at a time so that a true regeneration state can be determined. Instead of an event for the completion of an arrival time, there is an event for the completion of each stage of an arrival time. A state is accepted as a regeneration state only if all arrival times with the BE distribution are in the first stage. The simulation must handle service times similarly for classes with non-zero populations in the regeneration state. BE service times are sampled a stage at a time if and only if the corresponding class has non-zero population in the regeneration state. A state is accepted as a regeneration state only if all service times in progress which have the BE distribution are in the first stage.

A3.1.2. UNIFORM Distribution

The classical uniform distribution is one with uniform (positive) probability density over an interval (l, u) and zero density elsewhere. The uniform distribution provided by RESQ is a generalization of the classical form in that it allows several intervals instead of just one. (Note that we choose to exclude the interval end points in our definition of the classical uniform distribution. Similarly, our generalization of the classical form excludes the interval end points.) See Figure A3.2.

Each interval i, $i = 1,...,N$, is specified as a triple: l_i, u_i and p_i, representing the lower bound, the upper bound and the probability of the interval, respectively. (The probability of an interval is its width times its density.) The RESQ syntax is

$$\text{uniform}(l_1,u_1,p_1; \ldots ; l_N,u_N,p_N).$$

For example, the classical uniform distribution would be specified as

$$\text{uniform}(l,u,1)$$

The mean of the classical uniform distribution is given by

$$M = \frac{l+u}{2}$$

and the coefficient of variation is

$$C = \frac{u - l}{(l + u)\sqrt{3}}.$$

Alternatively, if we are given the mean and coefficient of variation,

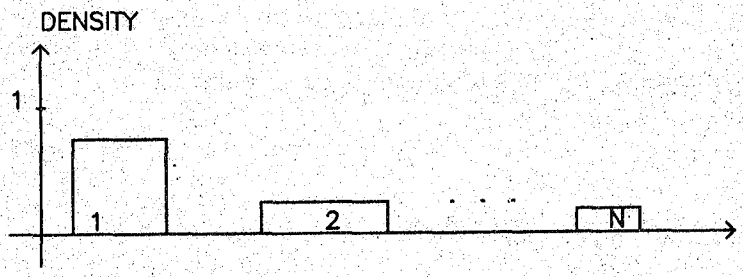

Figure A3.2 - UNIFORM Density Function

$$u = M(1 + C\sqrt{3})$$

and

$$l = 2M - u.$$

A3.1.3. STANDARD Distribution

In many circumstances one is satisfied by specifying a distribution by mean and coefficient of variation. RESQ includes a pragmatically chosen collection of distributions so specified. The syntax is

standard(M,C).

The distribution used will have mean M and coefficient of variation C where the specific form is chosen according to the value of C. If $C = 0$, then the constant value M is used. If $0 < C < .5$, then the classical uniform form is used. If $.5 \leq C < 1$, then the BE distribution is used with

$$K = \text{ceil}(C^{-2}),$$

$$p_1 = \frac{2KC^2 + K - 2 - \sqrt{K^2 + 4 - 4KC^2}}{(K-1)2(C^2+1)} \qquad (A3.3)$$

$$p_2 = \ldots = p_{K-1} = 0$$

and

$$m_1 = \ldots = m_K = M/(K - p_1(K-1)).$$

Here "ceil" is the ceiling function, i.e., it returns the next larger integer if its argument is not an integer and returns its argument otherwise. Note that this results in the Erlang distribution for $C = .5$, $\sqrt{3}$ and $\sqrt{2}$. If $C = 1$ the exponential distribution is used and if $C > 1$ the hyperexponential distribution specified is used with $K = 2$,

$$p_1 = C^2\left(1 - \sqrt{1 - \frac{2}{1+C^2}}\right), \qquad (A3.4)$$

$$m_1 = \frac{M}{1 + \sqrt{1 - \frac{2}{1+C^2}}} \qquad (A3.5)$$

and

$$m_2 = \frac{M}{1 - \sqrt{1 - \frac{2}{1+C^2}}}. \qquad (A3.6)$$

The discontinuity here, using the classical uniform distribution for small coefficient of variation and the BE distribution for larger coefficient of variation, is due to our general preference for the BE distribution tempered by the computational expense of using the BE for small coefficient of variation.

A3.1.4. DISCRETE Distribution

We have already discussed the DISCRETE distribution informally at the beginning of this appendix. The syntax is

$$\text{discrete}(v_1,p_1;\ \ldots\ ;\ v_N,p_N)$$

where p_i is the probability of value v_i.

In places where a discrete distribution is needed, e.g., for allocate nodes, it may be more convenient to use continuous distributions. If a distribution gives a fractional value for a value required to be an integer, the nearest integer is used. For example, if we want to specify the values 1 to 10 and 91 to 100 with equal probability, it would be more convenient to use

$$\text{uniform}(.5,10.5,.5;\ 90.5,100.5,.5)$$

than the explicit listing of all of these values with the discrete form. Note that since the uniform distribution excludes the end points of the intervals, the values 11 and 101 will not be produced by the above expression.

A3.1.5. Indirect Definition of Distributions

Where distributions are expected in RESQ dialogues, expressions containing distribution values may be used. For example, if we wanted the value from an exponential distribution with mean 10 shifted over 3 units, we could use

$$3+\text{standard}(10,1).$$

If we wanted the sum of a uniform and an exponential value, we could use something like

$$\text{uniform}(0,10,1)+\text{standard}(10,1).$$

Expressions not containing distribution keywords (BE, DISCRETE, STANDARD, UNIFORM) are taken to be the means of exponential distributions when such expressions are used for service times, work demands or arrival times. If we wanted a service time distribution to be the discrete distribution discussed in Section A3.1.4, (the values 1 to 10 and 91 to 100 with equal probability) then we could use

$$\text{ceil}(\text{uniform}(0,10,.5;\ 90,100,.5))$$

instead of the expression for this distribution suggested for allocate nodes. This expression could also be used for an allocate node and might be more clear in its intent than the expression which depends on rounding. However, this last expression would be less efficient in simulation run time than the one previously given. The reason is that the simulation has special cases for expressions consisting of a single distribution expression and its arguments. If those arguments are simulation independent, then the arguments are evaluated before simulation begins and the general expression code is avoided during simulation (for that expression).

Expressions containing distributions provide the opportunity to indirectly define distributions not directly provided by RESQ. For example, values from a geometric distribution (starting at one) with mean M can be obtained from the expression

$$\text{ceil}(\ln(\text{uniform}(0,1,1))/\ln(1-1/M))$$

where ln is the natural logarithm function.

A3.2. The USER Function

The USER function may be used to define distribution functions or other functions not directly available with RESQ and/or to provide RESQ with data from user defined files, e.g., for trace driven simulation. To do so, the user writes a PL/I function (with name USER) to be called whenever a RESQ expression contains a reference to USER. For example, the user might have

```
QUEUE:terminalsq
   TYPE:is
   CLASS LIST:terminals
      SERVICE TIMES:user(meantime;tq(memory))
```

or

```
SET NODES:set_leng
ASSIGNMENT LIST:jv(msg_lng)=user(mean_lng(jv(origin));clock)
```

In the dialogue, the USER function may have any number of arguments provided that at least one argument is given. These arguments are evaluated before the PL/I function is called and the values obtained are passed to the function as a vector. In addition to the arguments, four other values are passed to the PL/I function: (1) a seed to be used in generating random numbers, assuming the same generator is used as discussed in Section 12.3, (2) a pointer to the internal data structure used for the job causing the expression to be evaluated, (3) the internal number of the node causing the expression to be evaluated (the "to node" if this is a routing decision) and (4) the internal number of the queue (if any) to which that node belongs.

Only the function value returned by USER and the seed parameter are examined by the simulation program after the USER function returns to the calling procedure. The seed returned must be positive. If the seed returned is nonpositive, the simulation run will terminate with an error message. If the seed returned is different than the one supplied to the USER function, and the expression is used for service times, work demands or arrival times, the expression containing the USER function call will be treated as if it contained a RESQ distribution keyword (BE, DISCRETE, STANDARD, UNIFORM) whether it does or not.

If the user does not supply a USER function, the following version is used:

```
USER:
    PROC(ARGS,SEED,JOB,NODE,QUEUE) RETURNS(FLOAT BIN(53));
      DCL
         ARGS(*) FLOAT BIN(53),
         (SEED,NODE,QUEUE) FIXED BIN(31),
         JOB POINTER,
         FABORT ENTRY(CHAR(80));
      CALL FABORT('USER -- FUNCTION NOT DEFINED OR NOT LOADED');
    END;
```

The function FABORT causes performance measures to be determined as far as possible, then terminates the simulation with the error message which it received as its argument. At present there are no functions available to the user to take advantage of the job information. Func-

tions NODNAM and QUENAM are available to find the unqualified external name of a node or queue, respectively,

```
DCL
   NODNAM ENTRY(FIXED BIN(31)) RETURNS(CHAR(22) VARYING),
   QUENAM ENTRY(FIXED BIN(31)) RETURNS(CHAR(10) VARYING);
```

Functions NDQUAL and QUQUAL are available to find the qualification (names of invocations) of a node or queue, respectively,

```
DCL
   NDQUAL ENTRY(FIXED BIN(31)) RETURNS(CHAR(240) VARYING),
   QUQUAL ENTRY(FIXED BIN(31)) RETURNS(CHAR(240) VARYING);
```

After the USER PL/I function has been written, it should be compiled using the PL/I optimizing compiler, to produce a file USER TEXT. Then either the EVALT command (Section 13.2) or PL/I embedding (Section 14) should be used. The other procedures and functions we have just described (FABORT, NODNAM, etc.) will be automatically loaded with either approach.

A3.3. Status Functions

There are five functions which may be used in numeric expressions which indicate current status of the network. The functions have an argument specifying a node or queue name. When used in routing predicates, the argument is optional under the circumstances described with a specific function. These functions are

SA(queue name) - Servers Available. SA returns the number of servers currently available at an active queue, i.e., the number not in use. The queue name and parentheses may be omitted if the function is used in a routing predicate and the corresponding destination is a class of the queue.

TA(queue name) - Tokens Available. TA returns the number of tokens currently available at an passive queue, i.e., the number not in use. The queue name and parentheses may be omitted if the function is used in a routing predicate and the corresponding destination is an allocate node of the queue.

TH(queue name) - Tokens Held. TH returns the number of tokens of the specified passive queue held by the job causing the function to be invoked.

QL(node name) - Queue Length. QL returns the current number of jobs (counting both true jobs and job copies) at a class or an allocate node. The node name and parentheses may be omitted if the function is used in a routing predicate and the corresponding destination is the desired node.

TQ(queue name) - Total Queue. TQ returns the current number of jobs (counting both true jobs and job copies) at a queue. The queue name and parentheses may be omitted if the function is used in a routing predicate and the corresponding destination is a node of the desired queue.

RJ(node name) - Related Jobs. RJ returns the number of jobs related to the job causing the function to be invoked. If the node name is given, only jobs at that node are counted. Otherwise all of the job's relatives are counted.

(Related jobs are produced by fission nodes. See Section 8.) If the node name and parentheses are omitted, RJ returns the total number of jobs related to the job.

A3.4. The PRINT Function

In addition to the trace capabilities discussed in Section 12.4, the PRINT function may be used freely to follow the values of numeric expressions. The PRINT function takes a numeric expression as its sole argument and returns the value of its argument. For example, we might use

```
CREATE NODE LIST:c
   NUMBERS OF TOKENS TO CREATE:print(w+1)
```

or

```
SET NODES:set_leng
ASSIGNMENT LIST:jv(msg_lng)=print(user(mean_lng(jv(orig));clock))
```

Every time an expression using the PRINT function is evaluated, a line of the form

```
PRINT -- VALUE:    1.0100000E+02 ASSOCIATED WITH node
```

is produced at the at the terminal and in the RQ2PRNT file. If print is used in an expression for routing predicates or probabilities, the "associated with" node will be the destination being considered.

A3.5. Expression Evaluation

An expression is built up of primitive elements called factors. A factor may be an unsigned number, e.g.,

```
3   45.625   3.2E-10   4.356E+20   1.37E+02
```

(using *exactly* two digits for the exponent, if included) a parenthesized signed factor, e.g.,

```
(-1)
```

an identifier or global variable, e.g.,

```
disk   prob(2)   chn(1;*) newcyl
```

a job or chain variable,

```
jv(3)  cv(a_time_sc)
```

a distribution reference, e.g.,

```
standard(10,2)   discrete(1,.5;3,.5)  bE(1,0;1,1)
```

a status function call, e.g.,

```
ta(windowq)    th(windowq)  ql(class1)  rj
```

a parenthesized expression, e.g.,

```
(4+2)
```

a call to the USER defined function, e.g.,

```
user(ta(windowq);4+2;uniform(0,1,1))
```

a call to the PRINT function, e.g.,

```
print(ta(windowq))
```

or a numeric function call, e.g.,

```
min(1,2)      max(3,alpha)        ceil(10.3)      floor(9.99)
abs(beta)     exp(-3)             ln(exp(-3))
```

These numeric functions are evaluated using the corresponding functions provided by PL/I.

In SETUP and the RQ2COMP file, numerical values are generally treated as if they were single precision floating point, i.e., they are usually truncated to roughly six decimal digits, even if given more precisely by the user. Global variables, job variables, chain variables and temporaries used in expression evaluation are maintained as double precision floating point during the simulation, i.e., they have roughly 16 decimal digits of precision

Factors are combined with multiplying operators "*", "/" and "mod" to form terms, e.g.,

```
1*print(w)      alpha mod 2     jv(3)/jv(10)
```

(MOD is the modulo function, performed by the PL/I mod function.) A single factor may itself be considered a term if it is not to be used in an operation with a multiplying operator.

Terms are combined with adding operators "+" and "−" to form expressions, e.g.,

```
1*print(w)+5      alpha mod 2+1  5-jv(3)/jv(10)
```

Note that the multiplying operators are applied first before the adding operators. A single term may itself be considered an expression if it is not to be used in an operation with a adding operator. As suggested before, expressions may be parenthesized to force adding operators to be used before multiplying operators.

A3.6. Predicates (Boolean Expressions)

Predicates are used in routing definitions (Section 9). A predicate is a Boolean expression preceded by "if(" and followed by ")". A Boolean expression must evaluate to either T (true) or F (false).

The primitive elements of Boolean expressions are called Boolean factors. A Boolean factor may be a Boolean constant, e.g.,

```
T      F
```

a relational expression, e.g.,

```
v<2  j*k<=v  print(w)>j*k  abs(jv(0)) mod n>=print(w)  i=j  w¬=3.4
```

the logical negation of a Boolean factor, e.g.,

```
not v>3
```

or a predicate, e.g.,

```
if(1<=v and v<=10)
```

Note the use of "if" before the parenthesized Boolean expression. The following is incorrect:

```
if((1<=v and v<=10))
```

Boolean factors are combined with the logical "and" operation to form Boolean terms, e.g.,

```
1<=v and v<=10
```

A Boolean factor by itself may be considered a Boolean term if it is not to be used in an and operation.

Boolean terms are combined with the logical "or" operation to form Boolean expressions, e.g.,

```
1<=v and v<=10 or ta(windowq)=7
```

Note that the and is performed before the or. If the reverse order is desired, the predicate notation should be used, e.g.,

```
1<=v and if(v<=10 or ta(windowq)=7)
```

A Boolean term by itself may be considered a Boolean expression if it is not to be used in an or operation.

Note that all of the above Boolean expressions must be enclosed in "if(" and ")" before they can be used in routing definitions, e.g.,

```
host->link;if(1<=v and if(v<=10 or ta(windowq)=7))
```

APPENDIX 4 - BNF GRAMMAR

The following is a BNF grammar for the dialogue file language for RESQ. This grammar also applies to the interactive dialogue mode, but the interactive processor has additional restrictions, i.e., it excludes some portions of the grammar. In other words, this grammar shows some portions of the language which are accepted in dialogue files but are not present in interactive dialogue.

The non-terminal symbols are enclosed in angular brackets ("<",">"). The following metasymbols are used: "::=" "|" "[" "]" "{" "}" The square brackets and braces are extensions to the BNF notation to allow factoring and iteration, respectively. Strings in the braces may be iterated zero or more times, i.e., they need not appear at all. Two special non-terminals are used, <eol> for "end of line," and <empty> for the null string. A "line" will normally consist of one input record, but the special symbol "++" may be used to indicate concatenation, as discussed in Section 2. Wherever commas (",") are used as separators, one or more blanks may be used as well as or instead of a comma.

As usual, the grammar does not completely specifiy the syntax of the language. Certain sentences produced by this grammar are semantically invalid and must be rejected by the SETUP. These semantic restrictions are informally described in Sections 3-12.

Since this grammar is oriented toward dialogue files, a major omission has been made. Lines of the form "<prompt> : <eol>" are allowed in dialogue files where they would occur in interactive dialogue, but are left out of the grammar. This grammar also ignores the special queue types FCFS, PS, IS, etc. Many lines shown as required in this grammar are actually optional. As discussed in Appendix 2, singular forms of keywords may be substituted for plural forms when the singular form is not a separate keyword.

```
<model> ::=
    MODEL: <ident> <eol>
    METHOD: [ NUMERICAL | SIMULATION ] <eol>
    <numeric__param__dcl>
    <dist__param__dcl>
    <numeric__ident__dcl>
    <dist__ident__dcl>
    <global__var__dcl>
    <elem__array__dcl>
    <max__var__dcl>
    <queue__type__dcl>
    <queue__definitions>
    <set__definitions>
    <split__definitions>
    <fission__definitions>
    <fusion__definitions>
    <dummy__definitions>
    <network__temp__dcl>
    <network__temp__invocations>
    <chain__definitions>
    <method__dep__defs__1>
    <method__dep__defs__2>
    <method__dep__defs__3>
    END <eol>

<ident> ::= <letter> { <letter> | <digit> | $ | __ }
```

```
<letter> ::= A | B | ... | Z

<digit> ::= 0 | 1 | ... | 9

<expr> ::= <term> { <addop> <term> }

<term> ::= <factor> { <mulop> <factor> }

<factor> ::=
    <ident> | <array_ident> | <number> | ( <expr> ) |
    <sim_fcn_call> | <fcn_call> | <dist> | ( <sign> <factor> )

<array_ident> ::=
    <ident> ( <subscript_expr> { ; <subscript_expr> } )

<subscript_expr> ::=
    <expr> | *

<number> ::=
    [ <integer> | <integer> . <integer> | . <integer> | <integer> . ]
    [ E <sign> <integer> <integer> | <empty> ]

<integer> ::= <digit> { <digit> }

<sim_fcn_call> ::=
    TA | TA ( <queue_name> ) |
    SA | SA ( <queue_name> ) |
    TQ | TQ ( <queue_name> ) |
    QL | QL ( <node_name> ) |
    TH | TH ( <queue_name> ) |
    RJ | RJ ( <node_name> )

<queue_name> ::= { <invoc_ident> . } <id_or_arr_id>

<invoc_ident> ::= <ident> [ ( <expr> ) | <empty> ]

<node_name> ::= { <invoc_ident> . } <id_or_arr_id>

<fcn_call> ::= <fcn_ident> ( <expr> { , <expr> } )

<fcn_ident> ::= MIN | MAX | CEIL | FLOOR | ABS | PRINT | EXP | LN

<sign> ::= + | -

<mulop> ::= * | / | MOD

<addop> ::= + | -

<dist> ::=
    [ STANDARD | BE | UNIFORM | DISCRETE ]
        ( <expr> { [ , | ; ] <expr> } )

<numeric_param_dcl> ::=
    { NUMERIC PARAMETERS: <id_or_arr_id> { , <id_or_arr_id> } <eol> }
```

```
<id__or__arr__id> ::= <ident> | <array__ident>

<dist__param__dcl> ::=
    { DISTRIBUTION PARAMETERS: <id__or__arr__id> { , <id__or__arr__id> }
        <eol> }

<numeric__ident__dcl> ::=
    { NUMERIC IDENTIFIERS: <id__or__arr__id> { , <id__or__arr__id> } <eol>
    <id__or__arr__id> : <expr> { , <expr> } <eol> }
    { <id__or__arr__id> : <expr> { , <expr> } <eol> } }

<dist__ident__dcl> ::=
    { DISTRIBUTION IDENTIFIERS: <id__or__arr__id> { , <id__or__arr__id> } <eol>
    <id__or__arr__id> : <expr> { , <expr> } <eol> }
    { <id__or__arr__id> : <expr> { , <expr> } <eol> } }

<global__var__dcl> ::=
    { GLOBAL VARIABLES: <id__or__arr__id>
    { , <id__or__arr__id> } <eol>
    <id__or__arr__id> : <expr> { , <expr> } <eol> }
    { <id__or__arr__id> : <expr> { , <expr> } <eol> } }

<elem__array__dcl> ::=
    { CHAIN ARRAYS: <array__ident> { , <array__ident> } <eol> }
    { NODE ARRAYS: <array__ident> { , <array__ident> } <eol> }

<max__var__dcl> ::=
    [ MAX JV: <expr> <eol> | <empty> ]
    [ MAX CV: <expr> <eol> | <empty> ]

<queue__type__dcl> ::=
    { <queue__type__def> }
    [ QUEUE TYPE: <eol> | <empty> ]

<queue__type__def> ::=
    QUEUE TYPE: <ident> <eol>
    <numeric__param__dcl>
    <dist__param__dcl>
    <node__param__dcl>
    <queue__body>
    END OF QUEUE TYPE <ident> <eol>

<node__param__dcl> ::=
    { NODE PARAMETERS: <id__or__arr__id> { , <id__or__arr__id> } <eol> }

<chain__param__dcl> ::=
    CHAIN PARAMETERS: <id__or__arr__id> { , <id__or__arr__id> } <eol>
    { CHAIN PARAMETERS: <id__or__arr__id> { , <id__or__arr__id> <eol> }

<queue__definitions> ::= <queue__definition> { <queue__definition> }

<queue__definition> ::= QUEUE: <ident> <eol>
    [ <queue__body> | TYPE: <ident> <eol> <queue__type__invocation__params>
    | TYPE: <ident> : [ <expr> | <ident> | <array__ident> ]
```

```
    { ; [ <expr>  | <ident> | <array_ident> ] } <eol> ]

<queue_type_invocation_params> ::=
    <id_or_arr_id> : [ <expr> | <string> | <ident> | <array_ident> ] <eol>
    { <id_or_arr_id> : [ <expr> | <string> | <ident> | <array_ident> ] <eol> }

<queue_body> ::= <active_queue_body> | <passive_queue_body>

<active_queue_body> ::=
    TYPE: ACTIVE <eol>
    SERVERS: <expr> <eol>
    DSPL: [ FCFS | PRTYPR | LCFS | PS | SRTF | LRTF | PRTY ] <eol>
    [ PREEMPT DIST: <expr> <eol> | <empty> ]
    CLASS LIST: <id_or_arr_id> { , <id_or_arr_id> } <eol>
    WORK DEMANDS: <expr> { , <expr> } <eol>
    [ PRIORITIES: <expr> { , <expr> } <eol> | <empty> ]
    { CLASS LIST: <id_or_arr_id> { , <id_or_arr_id> } <eol>
    WORK DEMANDS: <expr> { , <expr> } <eol>
    [ PRIORITIES: <expr> { , <expr> } <eol> | <empty> ] }
    <server_def> { , <server_def> }

<server_def> ::=
    SERVER- <eol>
    RATES: <expr> { , <expr> } <eol>
    { RATES: <expr> { , <expr> } <eol> }
    [ { ACCEPTS: <id_or_arr_id> { , <id_or_arr_id> } <eol> }
    | ACCEPTS: ALL <eol> ]

<passive_queue_body> ::=
    TYPE: PASSIVE <eol>
    TOKENS: <expr> <eol>
    DSPL: [ FCFS | FF | PRTY | PRTYPR ] <eol>
    [ PREEMPT DIST: <expr> <eol> | <empty> ]
    { ALLOCATE NODE LIST: <id_or_arr_id> { , <id_or_arr_id> } <eol>
        <trans_alloc_creat_body> }
    { AND ALLOCATE NODE LIST: <id_or_array_id> { , <id_or_arr_id> }
        <eol> <trans_alloc_creat_body> }
    { OR ALLOCATE NODE LIST: <id_or_arr_id> { , <id_or_arr_id> } <eol>
        <trans_alloc_creat_body> }
    { TRANSFER NODE LIST: <id_or_arr_id> { , <id_or_arr_id> } <eol>
        <trans_alloc_creat_body> }
    { RELEASE NODE LIST: <id_or_arr_id> { , <id_or_arr_id> } <eol> }
    { DESTROY NODE LIST: <id_or_arr_id> { , <id_or_arr_id> } <eol> }
    { CREATE NODE LIST: <id_or_arr_id> { , <id_or_arr_id> } <eol>
        <trans_alloc_creat_body> }

<trans_alloc_creat_body> ::=
    NUMBERS OF TOKENS TO [ ALLOCATE | CREATE | TRANSFER ] :
        <expr> { , <expr> } <eol>
    [ PRIORITIES: <expr> { , <expr> } <eol> | <empty> ]

<set_definitions> ::=
    { SET NODES: <id_or_arr_id> { , <id_or_arr_id> } <eol>
    ASSIGNMENT LIST: <id_or_arr_id> = <expr>
```

```
        { , <id__or__arr__id> = <expr> } <eol> }

<split__definitions> ::=
    { SPLIT NODES: <id__or__arr__id> { , <id__or__arr__id> } <eol> }

<fission__definitions> ::=
    { FISSION NODES: <id__or__arr__id> { , <id__or__arr__id> } <eol> }

<fusion__definitions> ::=
    { FUSION NODES: <id__or__arr__id> { , <id__or__arr__id> } <eol> }

<dummy__definitions> ::=
    { DUMMY NODES: <id__or__arr__id> { , <id__or__arr__id> } <eol> }

<network__temp__dcl> ::= { <network__template> }

<network__template> ::=
    SUBMODEL: <ident> <eol>
    <numeric__param__dcl>
    <dist__param__dcl>
    <node__param__dcl>
    <chain__param__dcl>
    <numeric__ident__dcl>
    <dist__ident__dcl>
    <global__var__dcl>
    <elem__array__dcl>
    <queue__definitions>
    <set__definitions>
    <split__definitions>
    <fission__definitions>
    <fusion__definitions>
    <dummy__definitions>
    <network__temp__dcl>
    <network__temp__invocations>
    <chain__definitions>
    END <eol>

<network__temp__invocations> ::= { <network__temp__invocation> }

<network__temp__invocation> ::=
    INVOCATION: <id__or__arr__id> <eol>
    [ TYPE: <submodel__ident> : [ <expr> | <ident> | array__id> ]
        { ; [ <expr> | <ident> | <array__id> ] } <eol>
    | TYPE: <submodel__ident> <eol>
        <id__or__arr__id> : <expr> <eol>
        { <id__or__arr__id> : <expr> <eol> } ]

<chain__definitions> ::= { <routing__chain> }

<routing__chain> ::=
    CHAIN: <id__or__arr__id> <eol>
    TYPE: [ OPEN | CLOSED | EXTERNAL ] <eol>
    [ INPUT: <id__or__arr__id> <eol> | <empty> ]
    [ OUTPUT: <id__or__arr__id> <eol> | <empty> ]
```

```
    [ <source_definition> | POPULATION: <expr> | <empty> ]
    { : <routing_transition> <eol> }

<source_definition> ::=
    SOURCE LIST: <id_or_arr_id> { , <id_or_arr_id> } <eol>
    ARRIVAL TIMES: <expr> { , <expr> }

<routing_transition> ::=
    <from_part> -> <to_part> { -> <to_part> }

<from_part> ::= <node_name> { , <node_name> }

<to_part> ::=
    <node_name> { , <node_name> } ; <control_part>

<control_part> ::=
    FISSION | SPLIT |
    [ <expr> | <predicate> ] { , [ <expr> | <predicate> ] }

<predicate> ::= IF ( <Boolean_term> { OR <Boolean_term> } )

<Boolean_term> ::= <Boolean_factor> { AND <Boolean_factor> }

<Boolean_factor> ::=
    <Boolean_constant> | <expr> <relop> <expr>
    | <predicate> | NOT <Boolean_factor>

<Boolean_constant> ::= T | F

<relop> ::= = | ¬= | < | <= | > | >=

<method_dep_defs_1> ::=
    { QUEUES FOR QUEUEING TIME DIST: <queue_name> { , <queue_name> }
        <eol>
    VALUES: <expr> { , <expr> } <eol>
    { VALUES: <expr> { , <expr> } <eol> } }
    { QUEUES FOR QUEUE LENGTH DIST: <queue_name> { , <queue_name> }
        <eol>
    MAX VALUE: <expr> <eol>
    { MAX VALUE: <expr> <eol> } }
    { QUEUES FOR TOKEN USE DIST: <queue_name> { , <queue_name> } <eol>
    MAX VALUE: <expr> <eol>
    { MAX VALUE: <expr> <eol> } }
    { QUEUES FOR TOTAL TOKEN DIST: <queue_name> { , <queue_name> } <eol>
    MAX VALUE: <expr> <eol>
    { MAX VALUE: <expr> <eol> } }
    { NODES FOR QUEUEING TIME DIST: <node_name> { , <node_name> } <eol>
    VALUES: <expr> { , <expr> } <eol>
    { VALUES: <expr> { , <expr> } <eol> } }
    { NODES FOR QUEUE LENGTH DIST: <node_name> { , <node_name> } <eol>
    MAX VALUE: <expr> <eol>
    { MAX VALUE: <expr> <eol> } }
```

```
<method__dep__defs__2> ::=
    CONFIDENCE INTERVAL METHOD: [ NONE | REGENERATIVE | SPECTRAL
        | REPLICATIONS ] <eol>
    [ INITIAL | REGENERATION ] STATE DEFINITION- <eol>
    { CHAIN: <id__or__arr__id> <eol>
    NODE LIST: <node__name> { , <node__name> } <eol>
    REGEN POP: <expr> { , <expr> } <eol>
    INIT POP: <expr> { , <expr> } <eol> }
    CONFIDENCE LEVEL: <expr> <eol>
    [ NUMBERS OF REPLICATIONS: <expr> <eol>
    | SEQUENTIAL STOPPING RULE: [ NO | YES ] <eol>
        { QUEUES TO BE CHECKED: <queue__name> { , <queue__name> } <eol>
        MEASURES: [ QT | QTD | QL | QLD | TU | TUD | TT | TTD | TP | UT ]
        { [ QT | QTD | QL |QLD | TU | TUD | TT | TTS | TP | UT [ } <eol>
        ALLOWED WIDTHS: <number> { , <number> } <eol>
        EXTRA SAMPLING PERIODS: <integer> <eol> }
    | SEQUENTIAL STOPPING RULE: [ NO | YES ] <eol>
        { CONFIDENCE INTERVAL QUEUES: <queue__name> { , <queue__name> }
        <eol> MEASURES: [ QT | QTD ] { , [ QT | QTD ] } <eol>
        ALLOWED WIDTHS: <number> { , <number> } <eol> }
        { CONFIDENCE INTERVAL NODES: <node__name> { , <node__name> }
        <eol> MEASURES: [ QT | QTD ] { , [ QT | QTD ] } <eol>
        ALLOWED WIDTHS: <number> { , <number> } <eol>
        EXTRA SAMPLING PERIODS: <integer> <eol> }
    INITIAL PORTION DISCARDED: <expr> <eol> }
    [ SAMPLING PERIOD GUIDELINES- <eol> | RUN GUIDELINES- <eol>
    | REPLIC LIMITS- <eol> | RUN LIMITS- <eol> ]
    | INITIAL PERIOD LIMITS- <eol> ]
    SIMULATED TIME: <expr> <eol>
    CYCLES: <expr> <eol>
    EVENTS: <expr> <eol>
    QUEUES FOR DEPARTURE COUNTS: <queue__name> { , <queue__name> } <eol>
    DEPARTURES: <expr> { , <expr> } <eol>
    NODES FOR DEPARTURE COUNTS: <queue__name> { , <queue__name> } <eol>
    DEPARTURES: <expr> { , <expr> } <eol>
    LIMIT - CP SECONDS: <expr> <eol>
    SEED: <expr> <eol>

<method__dep__defs__3> ::=
    TRACE: [ NO | YES ] <eol>
    INITIALLY ON: [ NO | YES ] <eol>
    TURN TRACE ON- <eol>
    SIMULATED TIME: <expr> <eol>
    CYCLES: <expr> <eol>
    EVENTS: <expr> <eol>
    QUEUES FOR DEPARTURE COUNTS: <queue__name> { , <queue__name> } <eol>
    DEPARTURES: <expr> { , <expr> } <eol>
    NODES FOR DEPARTURE COUNTS: <queue__name> { , <queue__name> } <eol>
    DEPARTURES: <expr> { , <expr> } <eol>
    TURN TRACE OFF- <eol>
    SIMULATED TIME: <expr> <eol>
    CYCLES: <expr> <eol>
    EVENTS: <expr> <eol>
    QUEUES FOR DEPARTURE COUNTS: <queue__name> { , <queue__name> } <eol>
```

DEPARTURES: <expr> { , <expr> } <eol>
NODES FOR DEPARTURE COUNTS: <queue__name> { , <queue__name> } <eol>
DEPARTURES: <expr> { , <expr> } <eol>
JOB MOVEMENT: [YES | NO] <eol>
[QUEUES: [YES | NO] <eol>
| QUEUES: <queue__name> { , <queue__name> } <eol>
    {QUEUES: <queue__name> { , <queue__name> } <eol> }
EVENT HANDLING: [YES | NO] <eol>
EVENT LIST: [YES | NO] <eol>
SNAPSHOTS: [YES | NO] <eol>

APPENDIX 5 - SETUP ERROR MESSAGES

In addition to displaying error messages on the terminal, SETUP produces file RQ2LIST which contains the same error messages which were displayed at the terminal. Each error message is listed immediately following the statement which caused the error. (Erroneous lines given in interactive mode, and resulting error messages, will not appear in the dialogue or listing files.) The error messages from SETUP, followed by an explanation, are described below (in alphabetical order).

```
... HAS TOO MANY DIGITS
```

RESQ constants must have less than 11 digits. All but the first 10 digits will be ignored.

```
... IS AN IMPROPER CHAIN IDENTIFIER IN THIS SUBMODEL
```

The displayed identifier is either not a chain identifier or is an chain array which has already been defined in this (sub)model.

```
A LIST IN THIS LINE HAS TOO MANY ELEMENTS, BREAK IT
```

There are too many identifiers or expressions in the list. This problem can generally be circumvented by repeating the prompt and splitting the elements in the list among the repeated prompts.

```
ACTUAL NODE PARAMETER MISSING OR IN ERROR
```

During the invocation of a queue type, a formal node parameter of the queue type has been matched with an invalid node identifier.

```
ALL CLASSES IN A FCFS Q MUST HAVE SAME WDD WITH NUMERICAL SOLUTION
```

Only a single scalar expression can be given for the work demands of a FCFS queue in a model solved numerically. See Section 11 for other restrictions for numerical solutions.

```
ALL FORMAL PARAMETERS NOT MATCHED TO ACTUAL PARAMETERS
```

All the parameters of a queue type or submodel were not given a value in the invocation.

```
ALL NODES IN ACCEPTS LIST MUST BE TEMPLATE PARAMETERS
```

The classes listed in the ACCEPTS list of a server definition of a queue type must be parameters of the queue type.

```
ALL NODES IN CLASS LIST MUST BE QUEUE TEMPLATE PARAMETERS
```

All identifiers appearing in a CLASS LIST statement within a queue type must be node parameters of the queue type.

```
AN IDENTIFIER WITH RUNNING DIMENSIONS MUST BE ONLY ELEMENT IN LIST
```

Typically, an identifier with running dimensions is to be matched to a second identifier with running dimensions and thus must be the only element in the list a values matched to the second identifier.

ANALYSIS HAS BEEN SUSPENDED STARTING FROM THIS POINT

SETUP has detected an error which has forced the analysis to temporarily be suspended. SETUP will continue the analysis of the model as soon as possible.

ARRIVAL TIME DISTRIBUTION MISSING OR IN ERROR

The arrival statistics of source nodes in open chains must be declared.

AT LEAST ONE CHAIN PARAMETER REQUIRED IN SUBMODEL DEFINITION

All submodels must have one or more chain parameters.

AVAILABLE MEASURES ARE UT, TP, QL, QLD, QT, QTD, TU, TUD, TT, TTD

These are the only measures that can appear in the statement that begins MEASURES:

CHAIN TYPE CAN BE OPEN, CLOSED OR EXTERNAL (SUBMODELS)

Incorrect CHAIN TYPE: specified. External chains are legal only within a submodel definition.

CHAIN TYPE DCL MISSING OR IN ERROR

After a chain identifier is declared, there must be a statement beginning TYPE: , which declares the chain as open, closed or external.

COLON MISSING AFTER IDENTIFIER NAME

The syntax for assigning a value to an identifier is an identifier name followed by a colon followed by an expression.

COLON MISSING AFTER INCLUDE

The correct syntax is INCLUDE: followed by a file name. See Section 2.

COLON MISSING IN MODEL NAME DECLARATION

Correct syntax is MODEL: followed by the model name.

COMMENT TERMINATOR MISSING

A comment may not span multiple lines. Every comment must begin (/*) and end (*/) on the same line. Long comments can be included by consecutive comment lines.

DCL FOR EVENT HANDLING: YES OR NO, IS IN ERROR

This is one of the trace options.

DCL FOR EVENT LIST: YES OR NO, IS IN ERROR

This is one of the trace options.

DCL FOR JOB MOVEMENT ON OR OFF IS IN ERROR

This is one of the trace options. The syntax is JOB MOVEMENT: YES or NO.

DCL FOR QUEUES: YES, NO OR LIST IS IN ERROR

This is one of the trace options. The syntax is QUEUES: YES or NO or a list of queues.

DCL FOR SNAPSHOT: YES OR NO, IS IN ERROR

This is one of the trace options.

DECLARATION OF QUEUEING DISCIPLINE MISSING OR IN ERROR

All active queue definitions must have a statement which begins TYPE: in order to specify the queueing discipline at the active queue.

DECLARATION FOR TRACE INITIALLY ON OR OFF IS MISSING

This is one of the trace options and must be given when requesting trace output. The syntax is INITIALLY ON: YES or NO.

DECLARED DIMENSIONS OF IDENTIFIERS CANNOT BE RUNNING

The declared dimensions of identifiers and variables cannot contain an asterisk.

DEPARTURE COUNTS DECLARATION MISSING

Departure counts must be given for the queues specified.

DIMENSION FOR ... ARE UNDEFINED QUANTITIES

The dimension size for the displayed identifier contains an unknown identifier.

DIMENSIONALITY OF PARAMETER EXCEEDS MAXIMUM NUMBER ALLOWED

Numeric and distribution parameters can have up to 2 dimensions; node and chain parameters can have up to 1 dimension.

DIMENSIONS OF ACTUAL AND FORMAL PARAMETERS DO NOT AGREE

The dimensions of a submodel or queue template parameter do not agree with the dimensions of the expression which it is being assigned.

DIMENSIONS OF ACTUAL PARAMETER MUST BE RUNNING (*)

The value assigned to a formal parameter that is an array must be an identifier or an expression with the same number of running dimensions.

ELEMENT(S) IN CLASS AND VALUE LIST DISAGREE IN DIMENSIONS

Identifiers in the class list with running dimensions (*) must be matched with a value expression with at most one running dimension.

END OF FILE NOT REACHED. REMAINING LINES NOT PARSED

SETUP has processed the END statement for the model and considers the model complete. All lines after the END statement are not considered part of the model.

END OF QUEUE TYPE STATEMENT MISSING OR IN ERROR

The body of a queue type must be terminated by the statement END OF QUEUE TYPE "queue type name".

END OF SUBMODEL STATEMENT MISSING OR IN ERROR

The statement END OF SUBMODEL "submodel name" must appear at the end of every submodel.

ERROR DETECTED IN NODE DEFINITION

ERROR DETECTED IN QUEUE DEFINITION

ERROR IN VALUE EXPRESSION FOR PARAMETER

An incorrect expression has been given for the value of a queue template formal parameter.

EXPRESSION TABLE OVERFLOW. COMPILATION SUSPENDED. EXPRESSION OR ELEMENT VECTOR TABLE OVERFLOWS SIZE OF ...

SETUP has exhausted the available entries in one of its internal tables. The current size of the expression table or element vector table which has overflowed is given by (. . .). The problem can be rectified by increasing the values of EXPSIZ and ELVSIZ in the file SETUPD RQ2DAT. See Section 2.4 for a description of the SETUPD RQ2DAT file.

EXTERNAL CHAINS MUST BE DCL AS SUBMODEL PARAMETERS

The identifier given as a chain identifier for an external chain in a submodel must be a chain parameter of the submodel.

FCFS Q MUST HAVE EXPONENTIAL WORK DEMANDS WITH NUMERICAL SOLUTIONS

Only a single scalar arithmetic expression can be given as for the work demands of an FCFS queue in a model solved numerically. See Section 11 for other restrictions for numerical solutions.

FUSION/SPLIT NODES AND PREDICATES NOT ALLOWED IN NUMERICAL SOLUTION

The routing specification must use probabilities (between 0 and 1) when a model is solved numerically. See Section 11 for further restrictions for numerical solutions.

IDENTIFIER ... IS IMPROPERLY DEFINED

The specified name is not a valid chain, node or queue name.

IDENTIFIER NOT A STRING PARAMETER OF THIS QUEUE TEMPLATE

In order for an identifier to specify the queueing discipline of a queue type, the identifier must be a string parameter of the queue type.

IDENTIFIER NOT SUITABLE AS A VALID QUEUEING DISCIPLINE

The identifier is not a string parameter of the submodel or queue type.

ILLEGAL ARGUMENT(S) IN FUNCTION CALL

Either an incorrect expression of an incorrect number of expressions are contained between the parentheses following the function name.

IMPLICIT DUMMY NODES INVALID AS I/O NODES WITH NUMERICAL SOLUTION

Input and output nodes of an external chain must be previously defined classes of the submodel in a model with numerical solution. See Section 11 further restrictions for numerical solutions.

IMPROPER IDENTIFIER FOUND IN IDENTIFIER LIST

See Appendix 2 for discussion of legal RESQ names.

IMPROPER TYPE OF IDENTIFIER USED IN ARITHMETIC EXPRESSION

Only numeric identifiers can be used in all expressions. There are restrictions on the use of distribution, job, chain and global identifiers.

INCLUDE FILE NOT FOUND OR HAS INCORRECT RECORD FORMAT

The file to be included, with a file type of RQ2INP was not found on any accessed disk nor was found to be a member of any GLOBAL maclib. An included file must have fixed length records of length 80. See section 2 for a discussion of the INCLUDE statement.

INCORRECT ARITHMETIC EXPRESSION

See Appendix 3 for discussion of RESQ expressions.

INCORRECT CHAIN DEFINITION

See Section 9 for discussion of chains.

INCORRECT CONFIDENCE INTERVAL METHOD

The reply to CONFIDENCE INTERVAL METHOD: must be NONE, REGENERATIVE, REPLICATIONS or SPECTRAL.

INCORRECT DEFINITION OF INPUT OR OUTPUT NODES

See Section 10 for discussion of input and output synonyms.

INCORRECT DEFINITION OF NODES/QUEUES FOR CONFIDENCE INTERVALS

INCORRECT DEFINITION OF NODES OR QUEUES FOR DISTRIBUTIONS

An illegal node or queue name appears in the identifier list.

INCORRECT DEFINITION OF QUEUES OR NODES FOR DEPARTURE COUNTS

INCORRECT DEFINITION FOR THE SEQUENTIAL STOPPING RULE

See Section 12 for discussion of the sequential stopping rule.

INCORRECT DISCIPLINE CODE

An unknown queueing discipline has been specified. See Section 4 and 5 for discussion of queueing disciplines.

INCORRECT EXPRESSION FOR THE CONFIDENCE INTERVAL

A single arithmetic expression must specify the confidence interval; see Appendix 3 for discussion of RESQ expressions.

INCORRECT EXPRESSION FOR THE PREEMPTION DISTANCE

The preemption distance must be a single scalar arithmetic expression. See Appendix 3 for discussion of RESQ expressions.

INCORRECT EXPRESSION OR DISTRIBUTION EXPRESSION

See Appendix 3 for discussion of RESQ expressions.

INCORRECT INCLUDE STATEMENT. SPECIFIED FILE NOT INCLUDED.

The text in the file to be included will not be processed by SETUP. See the section on libraries for discussion of the INCLUDE statement.

INCORRECT INVOCATION

INCORRECT INVOCATION ARGUMENT

An incorrect expression has been given as a value for the prompted formal parameter of the submodel or queue being invoked.

INCORRECT JOB, CHAIN OR GLOBAL VARIABLE IDENTIFIER

See Sections 3 and 7 for discussion of the use of job, chain and global identifiers.

INCORRECT JOB OR CHAIN VARIABLE DECLARATION

The response to the MAX CV: or MAX JV: prompt must be a single arithmetic expression for the extent of the JV or CV vector.

INCORRECT JV SCALED LIST

Each element in the JV SCALED LIST should be YES, NO or an arithmetic expression.

INCORRECT METHOD DEPENDENT DEFINITION, ANALYSIS SUSPENDED

Due to an error in the method dependent information, SETUP cannot analyze the remainder of the model.

INCORRECT NESTING OF SUBMODELS, COMPILATION SUSPENDED

SETUP has found more END OF SUBMODEL statements then there are actual submodels.

INCORRECT NODE LIST

The list of nodes likely contains an invalid node name or a node which cannot be referenced in the current context.

INCORRECT NODE OR CHAIN IDENTIFIER

INCORRECT NUMBER OF ACTUAL PARAMETERS SPECIFIED

Expressions for parameter values were found when an end-of-line was expected. See Section 6 for discussion of matching formal parameters with actual values.

INCORRECT NUMBER OF WORK DEMANDS OR ARRIVAL TIMES

If there are n classes or source nodes, then there must be either 1 or n work demand or arrival times expressions.

INCORRECT OR ILLEGAL IDENTIFIER FOUND

See Appendix 2 for discussion of RESQ names.

INCORRECT OR IMPROPER NODE IDENTIFIER IN IDENTIFIER LIST

INCORRECT OR INVALID USER SUPPLIED PROMPT IN INPUT FILE

The prompt part of a statement (to the left of the colon) is not a valid RESQ prompt.

INCORRECT OR MISSING RELATIONAL OPERATOR

Valid relational operators are =, ¬=, >, >=, <, <= . See Appendix 3.

INCORRECT OR UNKNOWN SOLUTION METHOD

The solution method should be either numerical or simulation.

INCORRECT PARAMETER OR IDENTIFIER DECLARATION

INCORRECT PASSIVE QUEUE TEMPLATE DEFINITION

INCORRECT PREDICATE IN ROUTING DEFINITION

See Section 9 and Appendix 3 for discussion of routing predicates.

INCORRECT PRIORITY LIST

The priority list is discussed in Sections 4 and 5.

INCORRECT ROUTING TRANSITION

See Section 9 for discussion of routing.

INCORRECT SERVER DEFINITION

See Section 4 for discussion of server definition.

INCORRECT SET NODES DEFINITION

See Section 7 for discussion of set nodes.

INCORRECT SUBMODEL DECLARATION

INCORRECT SUBMODEL NESTING, END OF SUBMODEL ... ASSUMED

The end of the indicated submodel (...) was expected but not found.

INCORRECT TRACING DECLARATION

See section 12 for a discussion of the dialogue for simulation tracing.

INCORRECT USE OF FORMAL NODE PARAMETER OF A QUEUE TEMPLATE

The node parameter of a queue template can only be referred to within the body of the queue type.

INCORRECT WORK DEMANDS LIST

INITIAL POPULATION DECLARATION MISSING

In defining the initial state of a chain there must be a statement which begins INIT POP:

INPUT AND OUTPUT NODES CANNOT BE SUBMODEL PARAMETERS

The nodes specified in the INPUT: and OUTPUT: statements in an external chain definition cannot be parameters of the submodel.

INVALID IDENTIFIER IN CLASS LIST SPECIFICATION

See Appendix 2 for discussion of valid RESQ names.

INVALID INVOCATION IDENTIFIER QUALIFIES NODE, CHAIN OR QUEUE

One of the identifiers used to qualify the node, chain or queue name is either not a known invocation identifier or is an invocation identifier which cannot be referred to in the current context.

INVALID QUEUE NAME SPECIFIED

JOBS INITIALLY AT RELEASE, DESTROY, FUSION, SOURCE OR SINK NODES

The initial state description of a chain cannot have jobs initialized at any of these types of nodes.

JOB VARIABLES NOT ALLOWED IN ARRIVAL TIME DISTRIBUTIONS

MAXIMUM LEVEL OF SUBMODEL NESTING EXCEEDED. COMPILATION ENDS.

SETUP can handle up to 40 nested submodels at any one point in a model.

MAXIMUM NESTING OF INCLUDE STATEMENTS IS 10 LEVELS DEEP

SETUP can process at most 10 INCLUDE statements simultaneously. That is, a maximum of 10 non completed INCLUDE statements can be present during the text insertion required for an INCLUDE statement.

METHOD DECLARATION MISSING OR INCORRECT

After the MODEL: statement, there must be a statement which begins METHOD:.

MISSING "=" IN SET NODE SET-TO EXPRESSION

The correct syntax of an assignment list is a job, chain or global variable name, followed by an =, followed by an expression. See Section 7 for discussion of set nodes.

MISSING "IF" IN ROUTING PREDICATE

All routing predicates start with the word IF. See Section 9 for discussion of routing predicates.

MISSING LEFT OR RIGHT PARENTHESIS

NO ALLOCATE NODES HAVE BEEN DEFINED FOR THIS PASSIVE QUEUE

All passive queues must have at least one allocate node.

NODE AND CHAIN ARRAYS MUST BE ONE DIMENSIONAL

Node and chain arrays cannot be 2 dimensional.

NODE ARRAY REFERENCE IN NODE LIST MUST HAVE RUNNING (*) INDEX

The node in question has been declared as a node array parameter of the queue template and thus must have a running (*) index in the body of the queue type.

NODE PARAMETER IN LIST HAS NO CLASS ATTRIBUTES

A node parameter of the queue type was never defined as a class within the body of the queue type and thus cannot appear in the ACCEPTS list of the server definition.

NODE PARAMETER IS NOT USED IN THE BODY OF THE QUEUE TYPE

A node parameter was declared but never referenced with the queue type body.

NOT ALL IDENTIFIERS HAVE BEEN ASSIGNED AN INITIAL VALUE

When global, numeric and distribution identifiers are declared they must be assigned an initial value immediately after their declaration.

NUMBER OF RUNNING (*) DIMENSIONS DISAGREES WITH IDENT. DIMENSIONS

The number of running dimensions of identifiers in the expression does not agree with the number of dimensions of the identifier receiving the initial value.

NUMBER OF RUNNING (*) DIMENSION ON LEFT AND RIGHT OF "=" DISAGREE

In a set expression, the number of running dimensions of the job, chain or global variable must be the same as the number of running dimensions of the identifiers in the expression to the right of the "=".

NUMBER OF SET NODES AND SET-TO EXPRESSIONS DO NOT AGREE

If there are N set nodes in the identifier list then there must either one or N assignment lists. See Section 7 for discussion of set nodes.

NUMBER OF TOKENS MUST BE DECLARED

When defining a passive queue, there must be a statement beginning TOKENS: immediately following the TYPE: PASSIVE statement.

ONLY A SINGLE NODE/QUEUE FOR DEPARTURE COUNTS WITH SPECTRAL METHOD

ONLY A SINGLE SOURCE NODE IS ALLOWED WITH NUMERICAL SOLUTIONS

An open chain can only have a single source node in a model solved numerically. See Section 11 for further restrictions on numerical solutions.

ONLY CLASS, ALLOCATE AND FUSION NODES ALLOWED FOR QLD OR QTD

The only types of nodes at which queue length and queueing time distributions can be measured are class, allocate and fusion nodes.

ONLY CLASS, SOURCE AND SINK NODES ALLOWED WITH NUMERICAL SOLUTION

These are the only permissible node types if a model is to be solved numerically. See section 11 for a discussion of further restrictions on numerical solutions.

ONLY ONE DIMENSIONAL ARRAYS OF INVOCATION ARE ALLOWED

An invocation identifier can have at most one dimension.

ONLY VALID MEASURES FOR CONFIDENCE INTERVALS ARE QT AND QTD

This is true only with the spectral solution method.

PARAMETER DIMENSIONS MUST BE SPECIFIED AS RUNNING

The dimensions of array parameters must be declared as running (*).

PASSIVE QUEUES NOT ALLOWED WITH NUMERICAL SOLUTION METHODS

Only active queues are allowed in a model which is to be solved numerically. See Section 11 for further restrictions on numerical solutions.

POPULATIONS OR SOURCES DECLARATION IS MISSING

Open chain definitions must have a statement which begins SOURCE LIST:. Definitions for closed chains must have a statement which begins POPULATIONS:.

PREEMPTION DISTANCE NOT DECLARED

A queue with queueing discipline PRTYPR must have a statement which begins PREMPT DIST:. This statement must immediately follow the queueing discipline specification.

PRIORITY DECLARATION MISSING

A queue with a PRTY or PRTYPR queueing discipline must have a statement which begins PRIORITIES:

PRIORITY QUEUEING DISCIPLINES NOT ALLOWED WITH NUMERICAL SOLUTIONS

The only disciplines allowed in a model to be solved numerically are FCFS, LCFS, PS and IS. See Section 11 for further restrictions on numerical solutions.

QUEUE TYPE NAME MUST BE A VALID IDENTIFIER

An illegal identifier has been used to specify the queue name. See Appendix 2 for discussion of RESQ identifiers.

QUEUES TO BE CHECKED FOR SEQUENTIAL STOPPING RULE NOT DECLARED

After requesting the sequential stopping rule, there must be a line which begins QUEUES TO BE CHECKED:.

RANDOM NUMBER SEED DCL IS MISSING OR IN ERROR

The correct syntax is the prompt SEED: followed by a single arithmetic expression.

REGEN POPULATION ALLOWED ONLY AT ALLOCATE FUSION AND CLASS NODES

Each node listed in the regeneration state must be a previously defined allocate, fusion or class node.

REGENERATION POPULATION DECLARATION MISSING

After giving the node list in the regeneration state definition, there must be a line which begins REGEN POP:.

REGENERATION STATE NODE LIST CANNOT BE EMPTY

At least one node must be specified in the regeneration state node list.

REGENERATION STATE NODE LIST DECLARATION IS MISSING

After giving a chain identifier for the regeneration state definition, there must be a line which begins NODE LIST:.

REPLICATION LIMITS MISSING OR IN ERROR

After giving the number of replications, there must be the statement REPLIC LIMITS-.

ROUTING TABLE OVERFLOW. COMPILATION SUSPENDED. ROUTING TABLE OVERFLOWS SIZE OF ...

SETUP has exhausted the available entries in its internal routing table. The current size of the routing table is given by (...). This problem can be rectified by increasing the value of RTBSIZ in the file SETUPD RQ2DAT. See Section 2.4 for a description of the SETUPD RQ2DAT file.

SET TO DECLARATION MISSING IN SET NODE DEFINITION

Following the definition of the set node names, there must be a statement which begins SET TO: or ASSIGNMENT LIST:.

SIMULATION CP TIME IS MISSING OR IN ERROR

After defining the other limits or guidelines, a statement beginning LIMIT - CP SECONDS: must be given.

SINK NODES CAN BE PRESENT ONLY IN OPEN OR EXTERNAL CHAINS

A sink node cannot appear in the routing definition of a closed chain.

SINK NODES CANNOT BE USED IN CLOSED CHAINS

SPECIFIED IDENTIFIER NOT A PARAMETER OF THIS QUEUE TEMPLATE

Attempt made during invocation of queue type to assign a value to an identifier which is not a parameter of the invoked queue type.

SPLIT OR FISSION NODES MUST BELONG TO THE CURRENT SUBMODEL

A split or a fission node must a declared node of the (sub)model in which it is used in a routing definition.

STRING PARAMETERS NOT YET IMPLEMENTED FOR QUEUE DISCIPLINES

SYMBOL TABLE OVERFLOW. COMPILATION SUSPENDED. SYMBOL TABLE OVERFLOWS SIZE OF ...

SETUP has exhausted the available entries in its internal symbol table. The current size of the symbol table is given by (...). This problem can be rectified by increasing the value of SYMSIZ in the file SETUPD RQ2DAT. See section 2.4 for a description of the SETUPD RQ2DAT file.

TABLE DEFINITION FILE MISSING - DEFAULTS USED

SETUP was unable to find the file SETUPD RQ2DAT and has used default values for its internal table sizes. See section 2.4 for a description of the SETUPD RQ2DAT file.

THE ARRAY ... HAS BEEN DECLARED BUT NEVER USED AS A NODE OR A CLASS

A reference is being made to a node array that was declared but never defined as a class or other node.

THE CHAIN ... HAS NOT YET BEEN DEFINED

The indicated chain was used as an external chain in a submodel but has yet to be defined in the current (sub)model.

THE CONFIDENCE INTERVAL METHOD DECLARATION IS MISSING OR IN ERROR

After giving information about the nodes and queues for distribution measures, there must be a statement which begins CONFIDENCE INTERVAL METHOD:.

THE DIMENSIONS OF THE ARRAYS CANNOT BE UNDEFINED

Arrays must have dimension values that are known to SETUP.

THE IDENTIFIER BEGINNING ... HAS BEEN TRUNCATED TO 10 CHARACTERS

All RESQ identifier names must be 10 characters or less; any additional characters will be ignored.

THE IDENTIFIER BEGINNING ... IS IMPROPERLY DEFINED

The chain, node or queue name is incorrectly defined due to either an incorrect qualification or an improper node, queue or chain name.

THE IDENTIFIER ... IS AN UNKNOWN OR INCORRECTLY DEFINED QUEUE TYPE

Since the queue type name is undefined, it cannot be used in a queue type invocation.

THE IDENTIFIER ... IS AN IMPROPER NODE IDENTIFIER

The identifier shown is either an invalid identifier name, not a node identifier, or is a previously defined node identifier which cannot be referenced at this point in the (sub)model.

THE IDENTIFIER ... CANNOT BE UTILIZED IN THIS SUBMODEL

The displayed identifier has been declared outside the current scope and cannot be referenced at this point in the model.

THE IDENTIFIER ... HAS BEEN DECLARED TWICE

An identifier can only be declared once within a (sub)model. Model and queue type names can be used only once.

THE IDENTIFIER ... IS NOT A PARAMETER OF THIS SUBMODEL

An attempt is being made during an invocation to assign a value to an identifier that is not a parameter of the submodel being invoked.

THE IDENTIFIER ... WAS GIVEN A VALUE BUT NEVER DEFINED

The displayed identifier was not declared in this identifier declaration statement but an attempt is being made to assign it an initial value.

THE IDENTIFIER ... WAS ALREADY GIVEN A VALUE

The displayed identifier ... was already assigned an initial value.

THE LIST IN THE PREVIOUS STATEMENT CANNOT BE EMPTY

A null response is illegal to the prompt.

THE NODE ... IS IMPROPERLY DEFINED OR UNDECLARED

The node displayed has either yet to be defined or has been defined in another submodel but cannot be referenced here.

THE NUMBER OF VALUES DOES NOT MATCH THE NUMBER OF ...

If there are n identifiers then there must be either 1 or n values in the value list being matched to the identifier list.

THE ONLY ACCEPTABLE ANSWERS ARE "YES" AND "NO"

THE QUEUE ... IS IMPROPERLY DEFINED OR UNDECLARED

The queue displayed has either not be declared or was incorrectly declared.

THE PARAMETER ... HAS NOT YET BEEN ASSIGNED AN ACTUAL VALUE

The displayed parameter (...) of the submodel or queue type being invoked has not yet been given a value in this invocation.

THE STRING "RUN LIMITS -" IS MISSING

If the confidence interval method is NONE, then after the initial state definition is given, there must be a line "RUN LIMITS -".

THE STRING "RUN PERIOD GUIDELINES- " IS MISSING

When using the regenerative confidence interval method without the sequential stopping rule, there must be a line "RUN PERIOD GUIDELINES- ".

THE STRING "SAMPLING PERIOD GUIDELINES -" IS MISSING

When using the regenerative confidence interval method and the sequential stopping rule, there must be a line "SAMPLING PERIOD GUIDELINES -".

THE STRING "SEQUENTIAL STOPPING RULE: " IS MISSING

When using the regenerative confidence interval method, after specifying the confidence interval, there must be a line beginning SEQUENTIAL STOPPING RULE: .

THE SUBMODEL ... CANNOT BE INVOKED AT THIS LEVEL

The submodel name shown (...) cannot be used in an invocation at this point in the (sub)model.

THE SUBMODEL ... HAS NOT BEEN DEFINED

The identifier shown (...) was never defined as a submodel and hence cannot be used in a submodel invocation.

TYPE DECLARATION IS MISSING IN INVOCATION

The statement beginning INVOCATION: must be immediately followed by a statement beginning TYPE: in order to declare the name of the submodel being invokes.

UNABLE TO PERFORM FILEDEF FOR INCLUDE FILE

SETUP was not able to perform a CMS FILEDEF for a file with the given name. See Section 2.3 for discussion of the files to be included.

UNDEFINED IDENTIFIER FOUND IN RELATIONAL EXPRESSION

UNKNOWN IDENTIFIER SPECIFIES QUEUEING DISCIPLINE

The only identifier that can specify a queueing discipline is a previously declared string parameter.

USE OF NESTED INVOCATION NAMES IS NOT ALLOWED

An invocation name cannot appear in an expression for the index of an array invocation identifier.

VALUE EXPRESSION CONTAINS UNDEFINED IDENTIFIERS

The expression contains identifiers not previously declared as identifiers or parameters of the model.

WARNING: EXTRANEOUS TOKEN(S) BEING SKIPPED UNTIL END OF LINE

There are more identifiers or expressions of the line than SETUP expected - these extraneous identifiers or expressions will be ignored. This warning is issued, for example, when two initial values are given on the same line for a scalar numeric identifier.

WARNING: IMPLICITLY DECLARED NODE CREATED

A node which has not previously been defined has been used. The new node is implictly declared to be a dummy node.

WARNING: LOGICAL LINE LENGTH EXCEEDED. INCREASE LINSIZ IN SETUPD

SETUP has overflowed its input buffer for storing an entire logical line. This problem can be solved by increasing the value if LINSIZ in the file SETUPD RQ2DAT. See Section 2.4 for a description of the file SETUPD RQ2DAT.

WARNING: OPEN CHAIN WITH NO SINK NODE IN CURRENT MODEL LEVEL

The sink node for this open chain chain must have already been declared in a portion of this chain defined in a previous submodel invocation.

WARNING: SOURCE NODES DEFINITION MISSING IN OPEN CHAIN

Source nodes for this open chain must have already been defined in a portion of this chain defined in a previous submodel invocation.

WARNING: THE NODE ... HAS BEEN IMPLICITLY DECLARED

A node has been used which has not been previously declared or used. The default type for an implicitly created node is the dummy type.

WARNING: " " IS AN UNDEFINED CHARACTER - INPUT IGNORED

The displayed character (in between quotes) is not a character recognized by SETUP. The input will be processed as if this character never occurred.

WORK DEMANDS MUST BE DECLARED

The work demand distribution must be defined for every class of every queue.

YES AND NO CANNOT BE SPECIFIED TOGETHER WITH QUEUE NAMES

An response to a QUEUES: tracing prompt that includes a list a queue names indicates that only the specified queues will be traced. Thus, yes/no cannot also be specified with individual queue names.

In addition to the above error messages, SETUP contains internal error messages which should never occur. All such internal error messages begin with the phrase: "RESQ INTERNAL ERROR: ".

APPENDIX 6 - EVAL ERROR MESSAGES

The error messages produced by the EVAL command come from the expansion processor or a solution component.

A6.1. Expansion Processor Messages

The following messages are given in alphabetical order. Many of the messages are the result of internal consistency checks and should not occur.

EXPRESSION INVALID OR NOT IMPLEMENTED

The evaluation of this expression has probably not been implemented yet.

EXPRESSION TABLE EXCEEDED

An invalid expression has been encountered when attempting to evaluate an entry in the expression table.

INVALID CODE

This message is caused by an invalid response to the WHAT prompt. The response could contain an incorrect performance measure, inconsistent response (e.g., poci or rtmbo), or a suffix which is not ci or bo.

INVALID DEPARTURE COUNT

An invalid expression was given for a queue or node departure count.

INVALID DISTRIBUTION

An incorrect work demand distribution was specified.

INVALID DISTRIBUTION PARAMETER VALUE

An incorrect distribution parameter value was specified.

INVALID ELEMENT NAME

The element name given to the WHAT prompt or to subroutine GTRSLT is not in the symbol table.

INVALID ELEMENT TYPE

The element name given to the WHAT prompt or to subroutine GTRSLT is not a queue or node.

INVALID EXPRESSION TABLE POINTER

An invalid expression has been found.

INVALID MODEL PARAMETER NAME

The parameter name given to subroutine STPARM was not a model parameter in the symbol table.

INVALID NODE NAME

An incorrect node name has been specified.

INVALID PERFORMANCE MEASURE CODE

The performance measure code given to subroutine GTRSLT was not a valid code.

INVALID QUEUE NAME

An incorrect queue name has been specified.

INVALID ROUTING STATEMENT

An incorrect routing statement has been specified.

INVALID SYMTB TYPE FOR PARAMETER

A parameter has an incorrect symbol table type. This is probably an internal RESQ problem.

NODE DEFINED IN MORE THAN 1 CHAIN

The same node name has been used in more than one chain.

NOT IMPLEMENTED

The solution method or an expression is not implemented yet.

NUMBER OF BRANCHES EXCEEDS RANGE

This message would be produced if a routing branch was encountered by the expansion program which was larger than the initial size determined. This is probably an internal RESQ problem.

NUMBER OF CHAINS EXCEEDS RANGE

This message would be produced if a chain was encountered by the expansion program which was larger than the initial size determined. This is probably an internal RESQ problem.

NUMBER OF NODES EXCEEDS RANGE

This message would be produced if a node was encountered by the expansion program which was larger than the initial size determined. This is probably an internal RESQ problem.

NUMBER OF QUEUES EXCEEDS RANGE

This message would be produced if a queue was encountered by the expansion program which was larger than the initial size determined. This is probably an internal RESQ problem.

PARAMETER NAME NOT A VECTOR

A vector value has been specified for a scalar parameter in subroutine STPRMV.

PARAMETER VALUE CAN NOT BE NULL

A null value has been specified for a numeric parameter value.

QUALIFIED ROUTING NODES NOT IMPLEMENTED

Routing statements of the form inv1.node1->inv2.node2 are not implemented.

SIZE OF IEXPTB EXCEEDED

The size of the expression table for numeric parameter values has been exceeded. This is an internal RESQ problem.

WARNING - INIT. POP ¬= CLOSED CHAIN POP

The initial population specified is not equal to the closed chain population.

WARNING - NODE NOT BRANCHED FROM:

The named node is branched to but not from.

WARNING - NODE NOT BRANCHED TO:

The named node is branched from but not to.

WARNING - NODE NOT IN ROUTING:

The named node is defined, but not in the routing.

WARNING - PROBABILITIES DO NOT SUM TO 1

The probabilities out of a node do not sum to one.

WARNING - SUBMODEL NOT INVOKED:

The named submodel is defined but not invoked.

A6.2. Numerical Solution Messages

The SETUP command and expansion processor do almost all of the error checking for numerically solved models. The only messages produced are

A NETWORK WITH ALL QUEUE DEPENDENT RATE QUEUES MUST

```
HAVE AT LEAST ONE CHAIN THAT VISITS ALL QUEUES.
NUMERICAL SOLUTION NOT IMPLEMENTED FOR THIS NETWORK.
```

The implementation does not handle networks without this characteristic.

```
QUEUE q IS NOT CONNECTED TO FIXED RATE SUBNETWORK.
NUMERICAL SOLUTION NOT IMPLEMENTED FOR THIS NETWORK.
```

The implementation does not handle networks without this characteristic.

```
SOLUTION INFEASIBLE.  QUEUE q IS SATURATED
```

This message only occurs with networks with open chains. The arrival times are such that jobs arrive at queue "q" faster than they can be served.

```
SOLUTION NOT PERFORMED.  TOO MANY QUEUE DEPENDENT RATE QUEUES.
```

See discussion of RESQ2 NUMERD in Section 13.3.

A6.3. Simulation Messages

All simulation error messages begin with the name of the routine producing the message. The following list is given in alphabetical order. Many of the messages result from internal consistency checks and should not occur. The discussion below will focus on messages that are likely to occur and require further explanation. *For messages of the form "... STORAGE FULL" see also the discussion file RESQ APLMBD in Section 13.3.* Lower case characters are used to represent model specific information. i and j are used for integer values, x is used for floating point values, "ident" is used for an identifier, "node" is used for node names and "queue" is used for queue names.

ADEVNT adds events to the event list

```
ADEVNT -- EVENT LIST STORAGE FULL
ADEVNT -- NEW EVENT TIME BEFORE CLOCK
ADVENT -- PSEUDO EVENT AT FUTURE TIME
```

ALLCTE handles "plain" allocate nodes

```
ALLCTE -- ETPTR( i )= j
ALLCTE -- ETPTR( i )= j
ALLCTE -- JOB ALREADY HOLDS TOKENS OF queue
```

A job holding tokens at a given queue may not request additional tokens at that queue.

```
ALLCTE -- NET( node )= i
ALLCTE -- NP( node )= i
ALLCTE -- QD( queue ) NOT IMPLEMENTED
ALLCTE -- QUEUE queue NOT PASSIVE
ALLCTE -- TOKEN AMOUNT i AT node
```

Number of tokens requested must be positive

ALLTKN is used by ALLCTE and other passive queue routines.

```
ALLTKN -- JOB STORAGE AREA FULL
```

ANDOR handles AND and OR allocate nodes

```
ANDOR -- ETPTR( i )= j
ANDOR -- JOB ALREADY HOLDS TOKENS OF queue
```

A job holding tokens at a given queue may not request additional tokens at that queue.

```
ANDOR -- JOB DATA STORAGE FULL
ANDOR -- JOB STORAGE FULL
ANDOR -- JOB WITH OUTSTANDING PSEUDOS AT AND-OR NODE
ANDOR -- NET( n )= i
ANDOR -- TOKEN AMOUNT i AT node
```

Number of tokens requested must be positive

APLOMB is responsible for initializing variables for each run or replication.

```
APLOMB -- AREA STORAGE FULL
APLOMB -- ATTEMPT TO USE EXPERIMENTAL C.I. METHOD
APLOMB -- ETPTR( i )= j
APLOMB -- INVALID INITIAL PORTION DISCARDED
APLOMB -- INVALID JV SCALING VALUE
APLOMB -- JOB DATA STORAGE FULL
APLOMB -- JOB STORAGE FULL
APLOMB -- NAME(APLMBD)
APLOMB -- NEGATIVE INTERARRIVAL TIME AT node
APLOMB -- NET( i )= j
APLOMB -- NO NODE FOR GV INIT
APLOMB -- VALUE( ident )= i
```

AQTRAC is used for tracing active queues.

```
AQTRAC -- QUEUE queue LIST FAULTY
```

ARRIVE handles routing of jobs from node to node.

```
ARRIVE -- DESTINATION UNDEFINED
ARRIVE -- ETPTR( i )= j
ARRIVE -- INVALID INDICATOR P= i
ARRIVE -- JOB WITH RELATIVES AT SINK
ARRIVE -- NO DESTINATION CHOSEN. JUST LEFT NODE node
```

Probabilities do not sum to 1 and/or no true predicates.

```
ARRIVE -- NODE node NOT DEFINED
ARRIVE -- NULL JOB
ARRIVE -- RET( i )= j
ARRIVE -- TRACE STRING TOO LONG
ARRIVE -- UNDEFINED NODE TYPE, NODE= node
```

CHECK checks whether system is in regeneration state.

```
CHECK -- UNDEFINED C.I. METHOD i
```

COMPLT handles completions of service times at active queues.

```
COMPLT -- DSPL= i
COMPLT -- JOB job NOT IN QUEUE queue
COMPLT -- QUEUE queue DEFINITION NOT IMPLEMENTED
COMPLT -- QUEUE queue IS PASSIVE
COMPLT -- ZERO RATE NOT ALLOWED -- QUEUE queue LENGTH i
```

Expression for service rate at given length is not positive.

CREATE handles create nodes.

```
CREATE -- ETPTR( i )= j
CREATE -- NET( n )= i
CREATE -- TOKEN AMOUNT i AT node
```

Number of tokens created must be non-negative.

FISSN handles fission nodes.

```
FISSN -- JOB DATA STORAGE FULL
FISSN -- JOB STORAGE FULL
```

FUSN handles fusion nodes.

```
FUSN -- FISSION AND FUSION NODES NOT PAIRED
```

Relatives other than immediate family at the same fusion node.

GRLERL determines bE parameters for standard distribution.

```
GRLERL -- COVR= x
GRLERL -- MEAN= x
```

NEXPR evaluates numeric expressions.

```
NEXPR -- CAN'T FIND EXPRESSION FOR ident
NEXPR -- CV SUBSCRIPT i OUT OF RANGE
NEXPR -- ETPTR( i )= j
NEXPR -- ETPTR( i )= j
NEXPR -- EXPRESSION INVALID
NEXPR -- EXPRESSION INVALID OR NOT IMPLEMENTED
NEXPR -- EXPRESSION TABLE EXCEEDED
NEXPR -- FIXEDOVERFLOW
NEXPR -- ident SUBSCRIPT i OUT OF RANGE AT node
NEXPR -- INVALID EXPRESSION AT node
NEXPR -- INVALID NODE FOR QL
```

The QL status function applies only to classes and allocates.

```
NEXPR -- INVALID NODE NUMBER
NEXPR -- INVALID QUEUE FOR SA
```

The SA status function applies only to active queues.

```
NEXPR -- INVALID QUEUE FOR TA
```

The TA status function applies only to passive queues.

```
NEXPR -- INVALID QUEUE FOR TQ
NEXPR -- INVALID QUEUE NUMBER
NEXPR -- JV SUBSCRIPT i OUT OF RANGE
NEXPR -- NODE NAME EXPECTED AT node
NEXPR -- OVERFLOW
NEXPR -- SYMTB( i ).DIM_1= j
NEXPR -- SYMTB( i ).DIM_2= j
NEXPR -- TINDX OR VALUE( i )= j
NEXPR -- USER FUNCTION MUST HAVE AT LEAST ONE ARGUMENT
NEXPR -- USER FUNCTION RETURNS BAD SEED
```

The seed must remain positive after call to user defined procedure.

```
NEXPR -- VALUE( i )= J
NEXPR -- ZERODIVIDE
```

PASSIVE handles passive queue pseudo events (Appendix 7).

```
PASSIVE -- UNMATCHED NUMBER OF JOBS AND QUEUES
```

PEXPR handles evaluation of predicates.

```
PEXPR -- CAN'T FIND EXPRESSION FOR ident
PEXPR -- CV SUBSCRIPT i OUT OF RANGE
PEXPR -- ETPTR( i )= j
PEXPR -- EXPRESSION INVALID OR NOT IMPLEMENTED
PEXPR -- EXPRESSION TABLE EXCEEDED
PEXPR -- ident SUBSCRIPT i OUT OF RANGE AT node
PEXPR -- INVALID EXPRESSION AT
PEXPR -- INVALID EXPRESSION AT node
PEXPR -- JV SUBSCRIPT i OUT OF RANGE
PEXPR -- NOT IMPLEMENTED
PEXPR -- SYMTB( i ).DIM_1= j
PEXPR -- SYMTB( i ).DIM_2= j
PEXPR -- TINDX OR VALUE( i )= j
PEXPR -- VALUE( i )= j
```

PQTRAC handles passive queue trace.

```
PQTRAC -- QUEUE queue LIST FAULTY
```

REMVEV cancels pending events which become invalid.

```
REMVEV -- ATTEMPT TO REMOVE PSEUDO OR PRTYPQ EVENT
```

SAMPLE obtains distribution samples not involving simulation dependent values (other than random number streams).

```
SAMPLE -- DIST. STAGE= i
SAMPLE -- DISTRIBUTION TYPE= i
SAMPLE -- TOO MANY STAGES -- TYPE i
```

SERARR handles arrivals at active queues.

```
SERARR -- CYCLIC DISCIPLINE, QUEUE queue
SERARR -- ETPTR( i )= j
SERARR -- F.F. WITH ACTIVE QUEUE queue
SERARR -- NEGATIVE SERVICE TIME AT node
SERARR -- NET( i )= j
SERARR -- NP( i )= j
SERARR -- QD( queue )= i
SERARR -- QUEUE queue DEFINITION NOT IMPLEMENTED
SERARR -- ZERO RATE NOT ALLOWED -- QUEUE queue LENGTH i
```

Service rate for given length not positive.

SETNOD handles set nodes.

```
SETNOD -- ATTEMPT TO CHANGE CLOCK
SETNOD -- ATTEMPT TO CHANGE CPSECONDS
SETNOD -- CV SUBSCRIPT i OUT OF RANGE
SETNOD -- ETPTR( i )= j
SETNOD -- EXPRESSION TABLE EXCEEDED
SETNOD -- INVALID EXPRESSION AT node
SETNOD -- JV SUBSCRIPT i OUT OF RANGE
SETNOD -- NET( node )= i
SETNOD -- SYMTB( i ).DIM_1= j
SETNOD -- SYMTB( i ).DIM_2= j
```

SMULAT is the central routine which removes events from the event list.

```
SMULAT -- APPARENT DEADLOCK (EVENT LIST EMPTY)
SMULAT -- ETPTR( i )= j
SMULAT -- JOB DATA STORAGE FULL
SMULAT -- JOB STORAGE FULL
SMULAT -- NEGATIVE INTERARRIVAL TIME AT node
SMULAT -- NET( i )= j
```

SNKFUS handles sinks and fusion nodes.

```
SNKFUS -- AND-OR QUEUE NOT FOUND
```

SPLIT handles split nodes.

```
SPLIT -- JOB DATA STORAGE FULL
SPLIT -- JOB STORAGE FULL
```

TRAN handles transfer nodes.

```
TRAN -- CHILD ALREADY HOLDS TOKENS OF queue
```

Transfer is not allowed if the recipient already holds tokens of the queue.

```
TRAN -- CORRECT COPY OF CHILD NOT FOUND
```

Child is attempting to transfer tokens which it does not hold.

```
TRAN -- CORRECT COPY OF PARENT NOT FOUND
```

Parent is attempting to transfer tokens which it does not hold.

```
TRAN -- ETPTR( i )= j
TRAN -- NET( n )= i
TRAN -- NU=i CHILD HOLDS j
```

Child is attempting to transfer less than all of its tokens

```
TRAN -- NU=i PARENT HOLDS j
```

Parent is attempting to transfer less than all of its tokens

```
TRAN -- PARENT ALREADY HOLDS TOKENS OF queue
```

Transfer is not allowed if the recipient already holds tokens of the queue.

USER is for user defined numeric functions (Appendix 3).

```
USER -- FUNCTION NOT DEFINED OR NOT LOADED
```

APPENDIX 7 - EVENT HANDLING

With models using passive queues, fission nodes and/or split nodes, one must be conscious of the likelihood of several jobs moving at the same simulated time, say because of the release of enough tokens for several jobs waiting at allocate nodes to each be allocated tokens. There are a number of rules applied to prevent difficulties in such situations, but difficulties can still arise. *It is up to the user to understand the rules and mechanisms to avoid possible difficulties with simultaneous events.* We first informally discuss the intent of the simulation event handling mechanism and then describe the mechanism itself.

A7.1. Simultaneous Job Movement

The intent of the mechanism is that:

1. Once a job begins to move, it will continue to move until (a) it reaches an active queue, (b) it reaches an AND allocate node, an OR allocate node or an allocate node for a PRTY passive queue, (c) it stops at an allocate node (e.g., because sufficient tokens are not available), (d) it stops at a fusion node or (e) it leaves the network.
2. Whenever tokens become available attempts to allocate tokens to waiting jobs will be deferred until all jobs able to move at the current simulated time stop moving, according to (1). Once all jobs have stopped moving by (1), if one or more jobs that had been waiting for tokens have potentially become able to move, an attempt is made to allocate tokens to those jobs. This is done for each passive queue, one at a time, in the order that the potential for movement of jobs was discovered. Jobs allocated tokens at one queue are allowed to move as far as possible according to (1) before the next queue is treated.
3. Any jobs which had been stopped (e.g., are waiting for tokens) and can proceed because of side effects of another job's behavior, (e.g., release of tokens) are allowed to move, one at a time, as far as possible according to (1) and (2). If there are several such jobs, they are handled in the order in which they became able to proceed. These jobs move before jobs are allowed to move because of completion of service time and/or arrival from a source, *even if the service or (inter-)arrival time ends at the current simulated time.* Note that a zero service time at a queue can be used as a buffer, to artificially stop a job's movement to let other jobs move.

The rules satisfactorily deal with most situations. However, there is a potential for problems with multiple PRTY passive queues. Consider Figure A7.1 and assume that node a belongs to one PRTY queue, nodes b and c belong to another PRTY queue and that node b has priority over node c. Suppose that a job arrives at node c and after that, but at the same simulated time, another job arrives at node a. Both jobs would be stopped and the job at node c would then be given a chance at token allocation before the job at node a had a chance at allocate

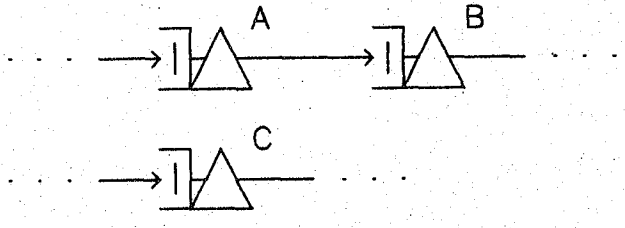

Figure A7.1 - Passive Queue "Race" Resolution

node b. If the job at node a were successful at node a and proceeded to node b, we could have the situation where both jobs had arrived at the queue at the same simulated time, but the job at the lower priority node got tokens and the job at the higher priority node did not. *It is up to the user to ensure that such problems do not occur.* One way to resolve this problem is by appropriate ordering of the allocate nodes. In the example, if node b were placed before node a, the problem would not occur. AND allocate nodes are also useful in avoiding problems such as this.

A7.2. Simulation Events

The simulation program has two classes of events, "pseudo events" and "real events." Only real events are counted in the simulation summaries produced by EVAL. Pseudo events always occur at the current simulated time and are intended to be transparent to the user except in the simultaneous job movement situations just discussed. Real events correspond to completion of service times and arrival times. (In models using the regenerative method, a real event may correspond to the completion of a stage of the service time rather than the entire time.) Once handling of an event begins, it is not interupted by scheduling of other events. Any pending pseudo events are handled before a real event is handled. Real events are handled in order of simulated time. In the case of real events at the same simulated time, the events are handled in the order they were scheduled.

There are two types of pseudo events, "pseudo arrivals" and "passive queue." Any pending pseudo arrival events are handled before a passive queue event. Among pseudo arrival events, events are handled in the order in which they were scheduled. Pseudo arrival events may be scheduled because of (1) initialization of jobs at the beginning of a run or replication, (2) generation of jobs by a split or fission node, or (3) the allocation of tokens to jobs by a passive queue pseudo event. Passive queue events may be scheduled because of (1) release of tokens, (2) creation of tokens, (3) arrival at a PRTY passive queue, (4) arrival at an AND allocate node, or (5) arrival at an OR allocate node. Among passive queue events, events are handled in the order in which they were scheduled.

Service completion events are scheduled because of a job beginning service at an active queue. Service completion events may be rescheduled because of preemption or changes in length at a processor sharing queue. Arrival time completion events are scheduled at the beginning of simulation and at the end of an arrival time. Arrival time events may be rescheduled or canceled because of changes to CV(0).

APPENDIX 8 - INSTALLATION

The RESQ distribution tape contains 17 files, including machine readable copies of this document and the RESQ Introduction and Examples document. (These document copies are formatted for printing on a line printer and do not contain the diagrams and some of the equations found in the standard paper copies.) After loading these files from tape to disk, the installer generates five additional module files using the RQ2MOD EXEC found on the tape. All 22 files together require roughly 6.5 million bytes of disk storage, e.g., roughly 14 cylinders of a 3350. However, one large file from the tape (COMPLIB TXTLIB) is not needed once the modules are generated. If this file and the two document files (also large files) are not retained on disk, then roughly 4.2 million bytes of disk storage, e.g., roughly 9 cylinders of a 3350, are required for the RESQ files. If additional conservation of disk space is desired, and the EVALT command and PL/I embedding are not to be used, then the other three TXTLIB files (EXPANSUB, MVASUB and APLOMB2) need not be retained on disk either, reducing the disk storage requirement to roughly 2.3 million bytes, e.g., roughly 5 cylinders of a 3350. The following discussion assumes (1) that the PL/I optimizing compiler is available on an accessed minidisk as PLILIB TXTLIB, (2) that 5.5 million bytes of disk storage (roughly 12 3350 cylinders) is, at least temporarily, available for at least the 20 files other than the document copies, (3) the disk for the RESQ files is accessed as the A disk, and (4) the tape is attached as virtual device 181.

Usually the installer will ask the machine operator to mount the tape and attach it to the installer's virtual machine. When the tape is ready, the message

```
TAPE 181 ATTACHED
```

should appear on the terminal. The user may then issue the CMS TAPE LOAD command, which will read the 15 RESQ files on the tape prior to the first tape mark, e.g.,

```
tape load
LOADING...
SETUP    EXEC     A1
EVAL     EXEC     A1
EVALT    EXEC     A1
RPLOT    EXEC     A1
RESQ2    APLMBD   A1
RESQ2    NUMERD   A1
SETUPD   RQ2DAT   A1
STXTLIB  MODULE   A2
STACK    MODULE   A2
SMACLIB  MODULE   A2
COMPLIB  TXTLIB   A1
EXPANSUB TXTLIB   A1
MVASUB   TXTLIB   A1
APLOMB2  TXTLIB   A1
RQ2MOD   EXEC     A1
END-OF-FILE OR END-OF-TAPE
R;
```

If the TAPE LOAD command is issued again, the remaining two files on the tape, the document copies, will be loaded. (This second TAPE LOAD command is omitted if the document copies are not desired on disk.)

```
RESQ      INTRO    A1
RESQ      CMSGUIDE A1
END-OF-FILE OR END-OF-TAPE
R;
```

Then the CP DETACH command is issued to have the tape rewound and detached from the installer's virtual machine:

```
detach 181
TAPE 181 DETACHED
R;
```

The RQ2MOD EXEC is now issued to generate the five MODULE files (COMPIL, EXPNDM, EXPWR1, EXPWRN, RAPLMB).

```
rq2mod
R;
erase load map
R;
```

(The RQ2MOD EXEC can be used to generate the modules one at a time. Issue "rq2mod ?" for an explanation of this option.) Now all RESQ files are in place on the disk. If any of the TXTLIB files are to be erased, they may be erased at this time.

To confirm the files have all been properly loaded and generated, issue the CMS LISTFILE command. Assuming all 17 files were loaded from the tape and that none of these files were subsequently erased, the output from LISTFILE might be

```
listfile (alloc
FILENAME FILETYPE FM FORMAT LRECL      RECS    BLOCKS
SETUP    EXEC     A1 F         80        77         8
EVAL     EXEC     A1 F         80       112        12
EVALT    EXEC     A1 F         80       115        12
RPLOT    EXEC     A1 F         80        19         2
RESQ2    APLMBD   A1 F         80         1         1
RESQ2    NUMERD   A1 F         80         1         1
SETUPD   RQ2DAT   A1 F         80         2         1
STXTLIB  MODULE   A2 V        272         2         1
STACK    MODULE   A2 V       1352         2         2
SMACLIB  MODULE   A2 V        272         2         1
COMPLIB  TXTLIB   A1 F         80     15086      1509
EXPANSUB TXTLIB   A1 F         80      8746       875
MVASUB   TXTLIB   A1 F         80      1274       128
APLOMB2  TXTLIB   A1 F         80     13994      1400
RQ2MOD   EXEC     A1 F         80        64         7
RESQ     INTRO    A1 V         95      8815       406
RESQ     CMSGUIDE A1 V         80     12970       687
COMPIL   MODULE   A1 V      65535         9       585
EXPNDM   MODULE   A1 V      65535         7       435
EXPWR1   MODULE   A1 V      65535         6       333
EXPWRN   MODULE   A1 V      65535         6       331
RAPLMB   MODULE   A1 V      65535        13       934
R;
```

The numbers of records and CMS blocks for the files may have changed slightly between this writing and the generation of the tape, so the installer should not expect to see exactly the figures shown above.

INDEX

G

H

I

J

L

M

N

O

P

Q

R

S