The Spring Simulation Testbed - V2
Design Document

Elizabeth Gene

May 1990

Contents

List of Figures

1 Overview

This document is in the directory:

`freal:~gene/testbed/doc/design/`

There is a Makefile to create it.

The Spring Simulation Testbed is divided into three modules:

1. Configuration Interface
2. Simulator
3. Animated Run-Time Display

This document describes in detail the design of the Simulator. The Configuration Interface and Animated Run-Time Display have relatively simple designs, and are described in the Programmer's Guide. I used SADT for the high level design of the Simulator, and an Ada/C hybrid to present the lower level design.

1.1 The Spring Simulation Testbed

The Spring Project has developed a number of heuristic scheduling algorithms. There is a need for a software testbed that can be used to evaluate these and other new algorithms for distributed, multiprocessor, hard real-time systems. The testbed will simulate a real-time operating system and provide an environment in which to experiment with the following items:

- Scheduling algorithms
- Run-time support mechanisms
- Hardware configurations

- Real-time loads

The system must be designed to facilitate replacement and modification of each module. In addition, the testbed must support various statistical analyses of the simulated system.

The testbed will be written partially in the simulation language SIMSCRIPT and partially in C. The mechanics of the simulation and the statistical analysis will be written in SIMSCRIPT, which provides good support for both. The modules that exist in the Spring kernel will be written in C, to facilitate replacement by other researchers. The data structures will also be in C, so the design of the abstract data types is crucial.

In the Spring project, we view a real-time application as a collection of application tasks that must each be **guaranteed** to finish execution by its deadline. The notions of guarantee and predictability are integral to the Spring methodology. Nothing that introduces uncertainty about the timing of tasks can be allowed. The kernel consists of processes, or system tasks, to support this methodology, including schedulers and dispatchers. In addition, the testbed contains task generators to simulate the arrival of tasks to the system. Tasks can be either periodic or non-periodic, but the management of periodic tasks has not yet been fully worked out. The task life cycle is as follows:

- A task is generated and placed in the *candidate queue.*
- The *local scheduler* (often called *scheduler*) removes it and places it in its working area, the *scheduler queue.*
- Using task characteristics such as *deadline, worst case computation time,* and *resource requirements,* the scheduler attempts to *guarantee* the task (assign it a *scheduled start time (SST)* such that it will meet its deadline).
- The guarantee may succeed or fail:
  - Success - it is placed in the *system task table (STT).*

- Failure - it is placed in the *send out queue* to be dealt with by the *global scheduler*. It may then be sent to another node to be scheduled there.

- The *dispatcher* takes it from the STT and gives it to the *processor* to run.
- When finished, the task is removed from the system.

The system consists of a number of nodes, each containing one or more application processors, one or more system processors, a dispatcher for each processor, one or more STTs, a periodic task generator, a local scheduler, a global scheduler, and a communications subnet. The non-periodic task generator is a global process. The design is complicated by the fact that there may be multiple STTs on a node. Each STT may be shared by several dispatchers or be owned by a particular dispatcher. My design concentrates on a single node.

1.2 Modifications to the SADT Method

In my high level design, I did not adhere strictly to the SADT rules. My application consists of a number of cooperating independent processes, which SADT does not accommodate very well. I took the liberty of using the following modified version of SADT:

1. Modules that are active processes, (as opposed to callable subroutines), appear in ovals rather than in boxes to emphasize the difference. The set of these processes pretty much coincides with the important modules in the Spring kernel.
2. I have divided the arrow labels into four categories:
   - Data
   - Imperatives for processes
   - Conditional results
   - Flow of control

To make this difference clear, I placed asterisks before the imperatives and surrounded the conditionals with dashes. Unlabeled arrows represent sequential operations.

2 High Level Design - SADT

In this section I will go through the design diagrams one by one. The diagrams appear at the end of this section. An outline of the modules by node number follows:

```
Testbed Top Level Design (G1)
1 Get Input Data
2 Initialize System
3 Run Simulation (G2)
  31 Generate Tasks (G3)
     311 Non-Periodic Task Generator
     312 Periodic Task Generator
     313 Add to Candidate Queue
     314 Send/Service Signals (G5)
         3141 Send Signal
         3142 Set Signal
         3143 Check Processor Interrupt Bit
         3144 Check Signals
         3145 Poller
         3146 Set Processor Status to INTERRUTPED
     315 Remove from Candidate Queue
     316 Compute Dynamic Task Info
  32 Local Scheduler - Heuristic (G6)
     321 Calculate and Set Cutoff
     322 Copy from STT
     323 Add to Scheduler Queue
     324 Remove from Scheduler Queue
     325 Try to Guarantee Tasks
     326 Set Task Status to READY
```

327 Rebuild STT With New Tasks

328 Return to Original STT

33 Schedule Tasks Globally

34 Interrupt Handler (G7)

341 Decrement Number of Allowed Interrupts

342 Select Signal to Service

343 Determine Whether to Honor Request

344 Determine Whether to Resume Task

345 Set Processor Status

35 Add to STT

36 Dispatch and Execute Tasks (G4)

361 Dispatcher (G8)

3611 Get Task to Dispatch (G12)

36111 Copy from STT

36112 Check if Task is Runnable

36113 Set Dispatcher Status to IDLE

36114 Sleep

36115 Set Dispatcher Status to PENDING

36116 Set Task Status to PENDING

36117 Sleep

36118 Set Task Status to READY

36119 Set Dispatcher Status to BUSY

3612 Set Task Status to EXECUTING

3613 Set Dispatcher Status to EXECUTING

3614 Set Processor Status to BUSY

3615 Set Processor Interrupt Bit

3616 Sleep

3617 Set Processor Status to IDLE

3618 Collect Statistics

3619 Remove Dispatched Task (G13)

36191 Check Task Status

36192 Remove Task from System

36193 Set Task Status to READY

362 Processor (G9)

3621 Sleep

3622 Get Execution Time

3623 Simulate Task Execution

3624 Calculate and Set Remaining Execution Time

3625 Check Processor Status

3626 Invoke Interrupt Handler (G11)

36261 Call Interrupt Handler

36262 Check Processor Status

36263 Set Task Status to DONE

36264 Set Task Status to READY

36265 Set Processor Status to BUSY

3627 Set Task Status to DONE

2.1 Testbed Top Level Design

At this level there is a shell around the simulation of the Spring kernel, which is detailed in diagram G2.

2.1.1 Get Input Data

The user interface, which has not yet been designed, will be interactive and window-oriented. The user controls the configuration of the system through selection of the following:

- System clock tick size
- Simulation length
- Number of nodes
- For each node:
    1. Number of processors/dispatchers
    2. Number of STTs
    3. Dispatcher/STT assignments and type of dispatching (processor bound or independent)
    4. ID of system processor
    5. Signal (system task requests) usage
    6. Application task workload
    7. Algorithms/controlling parameters (periodicity, computation time, etc.) for
        - local scheduling
        - global scheduling
        - signal vector polling

  - resource reclaiming
  - guarantee for local scheduling
  - heuristic function for guarantee
  - signal selection
  - signal honoring
  - task resumption after signal handling

- Network topology
- Channel speed
- Channel protocol
- RFB size/timeout

2.1.2 Initialize System

System initialization consists of setting control parameters and creating active processes, such as the processors, dispatchers, task generators, and subnets. The initialization activates all the modules on the following diagrams, and results are produced continually throughout the simulation.

2.2 Run Simulation

This diagram shows the flow of tasks through the system, most of which was described in Section 1.1. The diagram shows periodic and non-periodic tasks both handled by the local scheduler. I have not yet implemented any periodic task scheduling. As a temporary measure, there are two periodic task generators for each node, one for the local scheduler and one for the global scheduler. I think that the proposed design provides the greatest flexibility. If we want to experiment with having the two types of tasks contend for the processor and other resources, or with scheduling them on different processors, there will be one place to change the code. The local scheduler may call different subroutines to handle the different task types, but where scheduling is done is well defined.

The *Add to STT* box shows how I chose to represent data structure access. Any routine with an ADT mechanism operates on global operating system data. The only access allowed is through the ADT routines. As mentioned previously, this is essential for the dual-language design of the system since the data must be accessed consistently by routines written in both C and SIMSCRIPT. When tasks are added to the STT, the relevant dispatcher(s) are awakened (if sleeping) if the first dispatchable task has changed. This ADT routine has two levels:

1. Upper level - knows about system policy
   - Knows which STT is appropriate for the task
   - Wakes the dispatcher
2. Lower level - knows about data structure implementation
   - Adds the task to the data structure

Since both policy and mechanism may change independently, it is important to keep them as separate as possible. I chose to have this routine do the notification of the dispatcher, rather than the scheduler (as in the Spring kernel), to allow different ways of adding tasks to the STT, such as through interrupts. I will go into more detail about the *interrupt handler* when I describe diagram G7 but notice that new tasks are produced that go into the STT for dispatching.

2.3 Generate Tasks

This diagram shows the beginning of the task life cycle. The *non-periodic task generator* and *periodic task generator* generate tasks at statistically determined intervals and sleep until the next generation time. They can be easily replaced by another source of tasks to exercise the system. When tasks are added to the candidate queue, a request is made for a non-periodic invocation of the local scheduler. This will be explained in more detail later. However the scheduler is run, it removes tasks from the candidate queue. Run time task information, such as *worst case computation time*, is computed and added to the task structure before the scheduler gets the task. This is done at the last possible moment so that all system status information at scheduling time is available. The task structure itself is to be accessed only through the task ADT to facilitate modification and dual language access.

2.4 Local Scheduler - Heuristic Scheduling

As mentioned previously, the local scheduler is a periodic system task. It may also be invoked as a non-periodic system task. The following description assumes there is one new task in the candidate queue, and we are doing heuristic scheduling with guarantee. Note that only tasks on the same STT contend for resources.

To ensure that the dispatcher has tasks to dispatch while allowing the scheduler to reschedule tasks (to fit in the new task(s)), we use a *cutoff point.* All tasks that are not scheduled to begin execution before the scheduler will finish execution are eligible for rescheduling. The cutoff point is set in the STT and the dispatcher will not dispatch tasks past it. All tasks past the cutoff point are copied from the STT, so the STT can be restored easily if the new task is not schedulable.

The tasks to be (re)scheduled are placed into the scheduler queue. The scheduler removes them one by one and tries to guarantee them. As mentioned above, if the tasks cannot be guaranteed the original STT is restored and the new task is eligible for global scheduling. Otherwise, the new task is *ready* to be dispatched, and the STT is rebuilt with the new task.

This is just one possible scheduling algorithm. One of the primary purposes of the Testbed is to experiment with different scheduling algorithms. The cutoff method does not have to be used, or may be modified and the guarantee routine is replaceable as a whole.

2.5 Dispatch and Execute Tasks

This is the simplest diagram in the design. The dispatcher, when awakened, wakes the processor and gives it a task to run, then goes to sleep. When the processor is finished, it goes to sleep and wakes the dispatcher. This cycle is repeated forever.

2.6 Dispatcher

The dispatcher is a process that consists of an infinite work-sleep-work loop. It can be sleeping in two states:

1. Trying to get a task to dispatch
2. After dispatching a task to the processor

Assuming the dispatcher is in the first of these two states, it will be awakened when a new task is put into the STT and do the following:

- Give the processor the task to run
- Set the task status to *executing*
- Set its own status to *executing*
- Set the processor status to *busy*
- Set the processor interrupt bit (more on this later)
- Wake the processor and go to sleep

When the processor is finished, it will go to sleep and wake the dispatcher. The dispatcher then resets the processor status, removes the task, and tries to repeat the cycle. After the task is removed, each dispatcher that meets the following two conditions is awakened:

1. It shares this dispatcher's STT
2. It is *pending*

The removed task may have finished early, and perhaps other dispatchers can now dispatch a task.

2.7 Get Task to Dispatch

This diagram details how the dispatcher gets the next task to dispatch. When awakened or after removing a task from the processor, the dispatcher copies the first task from the STT. If there is none, the dispatcher goes back to sleep. Otherwise, the task must be checked for runnability. Tasks might complete execution faster than their worst case computation time, so the dispatcher may try to start a task before its scheduled start time. For now, this is disallowed, and the dispatcher will sleep until the SST or until awakened by *Add to STT.* The *Check if Task is Runnable* module can be replaced by a resource reclaimingalgorithm, which checks to see if dispatching a task early will cause resource contention. Eventually, the dispatcher has a task to run, which is returned.

2.8 Remove Dispatched Task

When the processor wakes the dispatcher, it removes the task. This description is complicated by interrupts. If the task has really completed, its status will be *done* and it will be removed from the STT and the system after collecting task statistics. Otherwise, it is left where it is and its status is reset to *ready.*

2.9 Processor

The processor is the most important module that does not exist in the Spring kernel, where there is a real processor. When awakened, the processor gets the actual execution time of the task and *works* for that long, or until interrupted, then computes how long the task actually ran and sets the remaining computation time. If there was an interrupt, the interrupt handler is invoked. Otherwise, the task status is set to *done.* The processor then wakes the dispatcher and goes to sleep.

2.9.1 Simulate Task Execution

In simulation, everything happens in zero time unless explicitly programmed otherwise. In SIMSCRIPT, this takes the form of a *work* statement. There are two parts to simulating task execution:

1. *Working* for the actual execution time of the task

2. Executing the actual task code (if any)

The actual task code may execute *work* statements also. The tasks are set up to have an array of function pointers for actual code and arrays of time values for execution time. This is done so a task may be broken up internally into separate functions, each with its own execution time component. The time vectors are constructed as follows:

- The 0th spot must be the sum of the rest of the elements

- The n+1th spot is the time to be worked before the nth section of code

- The n+2th spot is the time to be worked after the nth section of code

Each task has a variable representing the number of code sections actually in use. Zero time values and NULL code pointers are allowed. For example, my implementation of the heuristic local scheduler has one code section, which is a pointer to the scheduler code, and the time vector is all zeros. The time is consumed internally - the scheduler calls a function to determine how long it needs to work and executes *work* statements.

2.10 Interrupts

The Spring kernel does not currently support interrupts. There is no use for interrupts as commonly supported in operating systems because they introduce uncertainty about timing, which goes against the Spring methodology. However, there still may be some use for a limited set of interrupts. The Testbed will support requests for non-periodic invocations of tasks, such as the scheduler or a crucial task that must be run immediately. The former is important so that tasks do not waste time in the candidate queue before being scheduled, the latter to avoid potential system failure. I will provide a mechanism for handling interrupts, the policy decisions are up to the user. The interrupt facility comes with the warning *use at your own risk,* since the guarantee of tasks can be jeopardized by preemption.

I will give an overview of interrupt generation and handling and then describe the design in detail.

- Each task has an integer value field that represents the number of times it can be interrupted. Its worst case computation time includes that many invocations of the *interrupt handler.*
- Each processor has an interrupt bit, turned on or off by the dispatcher after checking if the task *number of allowed interrupts* is greater than zero.
- An interrupt is requested by sending a signal of the appropriate type, which causes a bit to be set in the *interrupt vector.*
- There is a *poller,* that runs on the system processor at regular intervals and checks the interrupt vector. Its execution time is included in the worst case computation time of all system tasks.

- After a task is interrupted by the poller, the interrupt handler determines if the request will be honored. If so, the new task is placed at the front of the STT.

2.11 Send/Service Signals

The top half of this diagram shows how interrupts are generated. The bottom half shows the operation of the poller. If the processor is interruptible and there is at least one signal, the processor status is set to *interrupted* and it is interrupted. When the processor receives an interrupt, it invokes the interrupt handler, as shown in diagrams G9 and G11.

2.11.1 RunPoller

There is a SIMSCRIPT process RunPoller that generates a poller task at user-specified intervals. At generation points, the following actions are taken:

- A poller task is created and initialized.
- Its *number of allowed interrupts* is set depending on the processor status:
  - If *busy*, it is copied from the task in execution.
  - If *idle*, it is set to one.
- The poller task bypasses scheduling and is sent to the head of the system processor's STT.
- If the processor is *busy*, we want to really interrupt it, but not have it execute the normal interrupt handler, since poller interrupts are always accepted. So, the interrupt handler algorithm is set to a function that only sets the processor status to *resume task after interrupt,* and the processor is actually interrupted. Otherwise, the poller task is just left on the STT for the dispatcher.

2.11.2 Poller

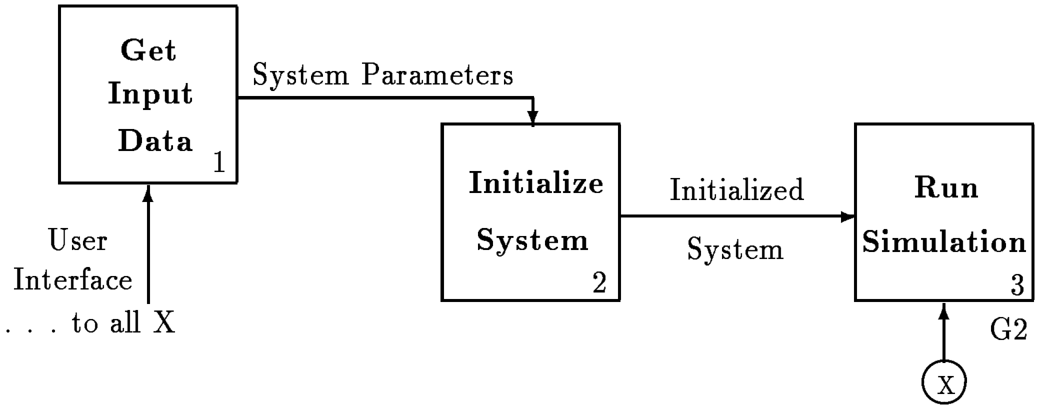
When the poller actually runs, the processor has already been interrupted to run it. If there is to be an interrupt, it sets the interrupt handler algorithm to the full handler, and calls the interrupt routine with a flag set so it will only set the processor status to *interrupted.* Then, when the processor is done simulating the poller execution, it will call the interrupt handler.

2.12 Invoke Interrupt Handler

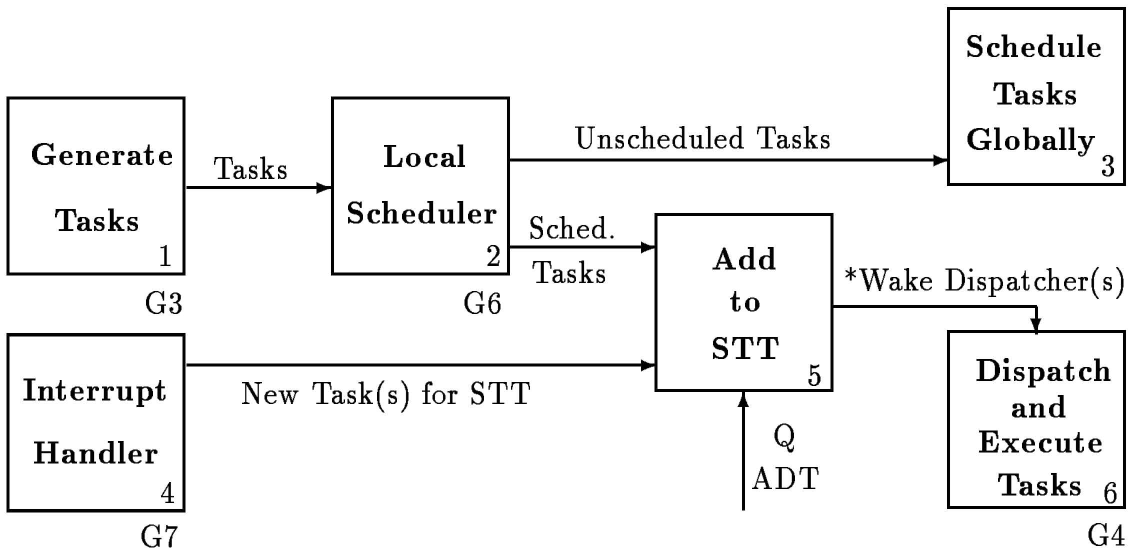
This is part of the processor code. when interrupted, the processor calls the interrupt handler. Upon return, it checks its own status to see if the interrupted task will ever be resumed. If so, the task status is set to *ready*, otherwise to *done.* The processor resets its own status to *busy,* and continues as if the task had terminated normally.

2.13 Interrupt Handler

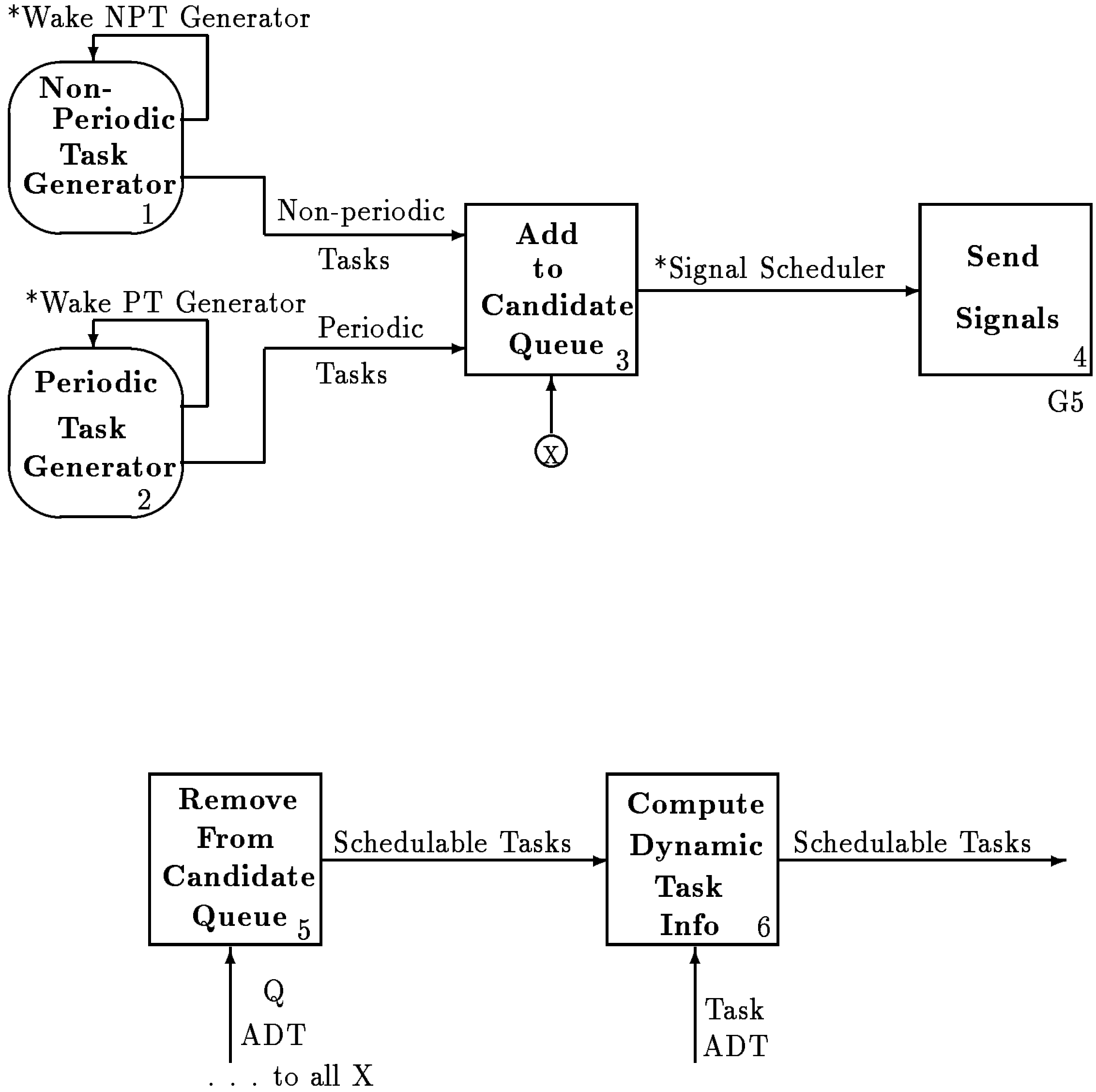
The interrupt handler first decrements the task *number of allowed interrupts.* The next three modules are user replaceable policy. If there is more that one signal pending, one must be selected. Then the interrupt handler must determine whether or not to honor the request, and whether or not to resume the task. The former decision results in a new task on top of the STT or not, the latter in a change in processor status. The interrupt handler then sets the processor status accordingly and returns.



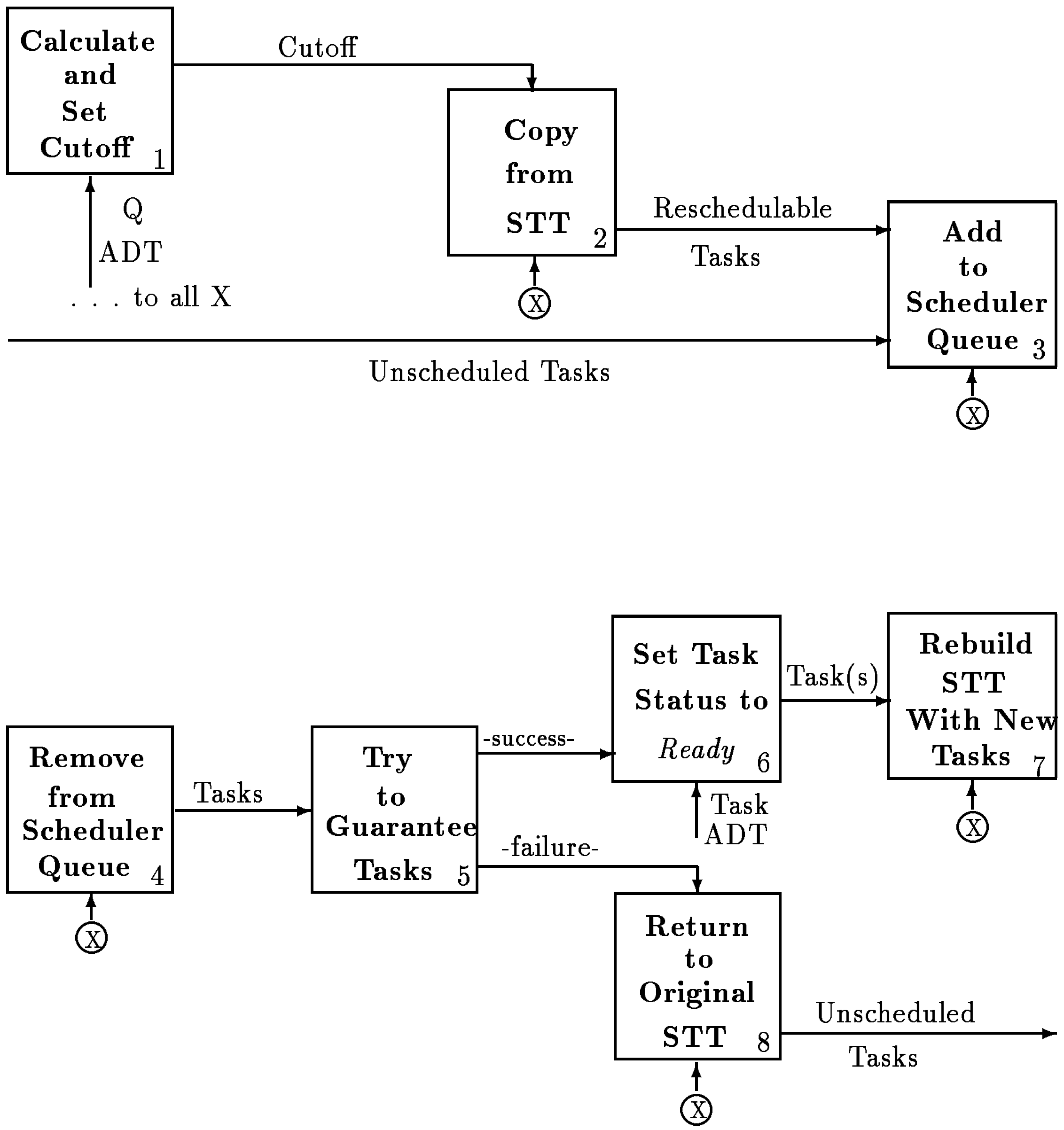
NODE: 0	TITLE: Testbed Top Level Design	NUMBER: G1



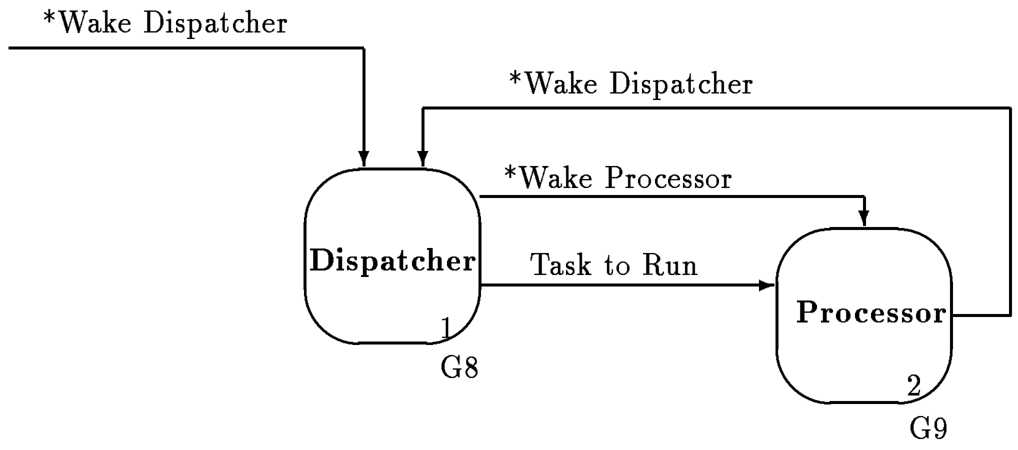
NODE: 3	TITLE: Run Simulation	NUMBER: G2



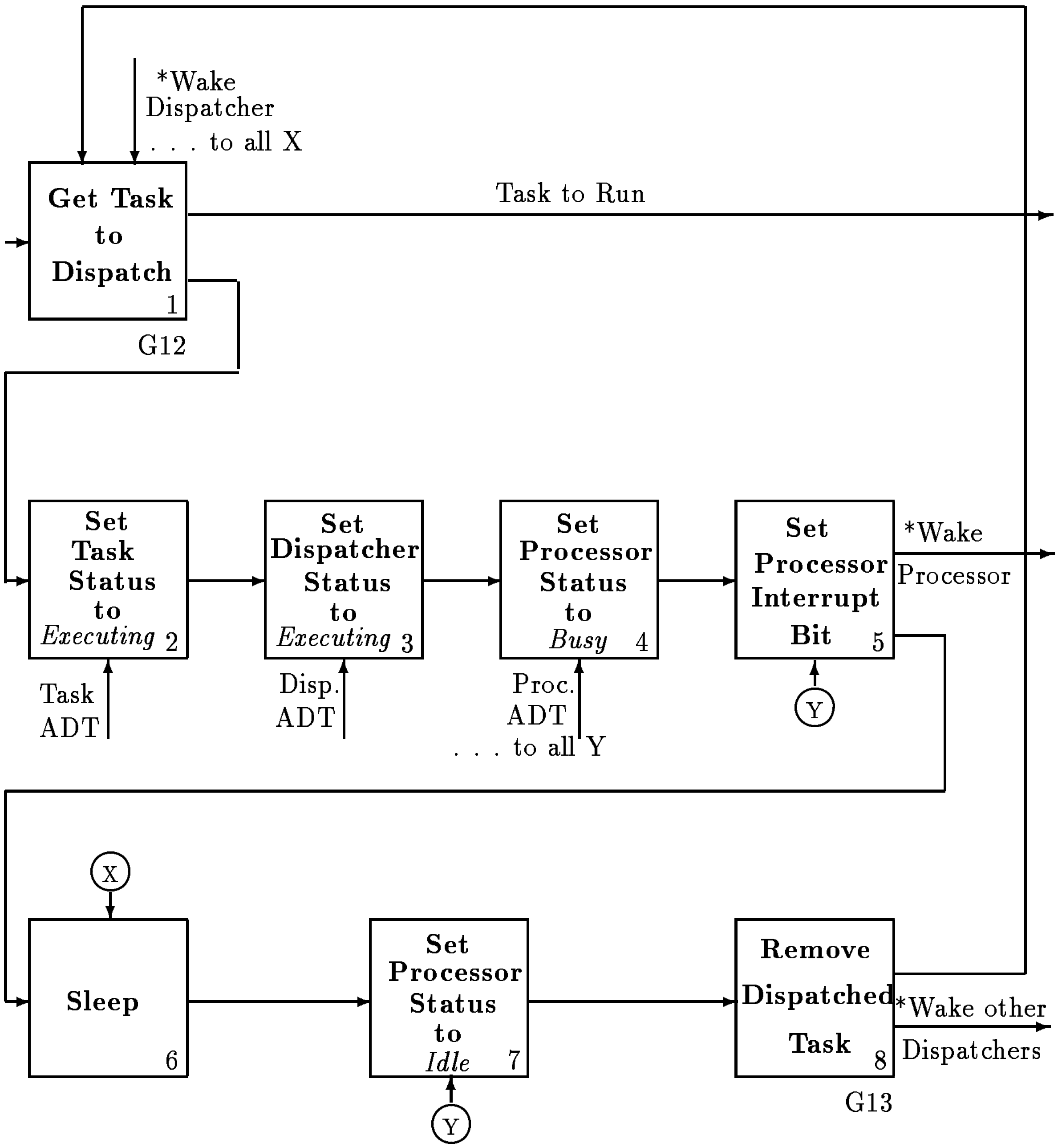
NODE: 31	TITLE: Generate Tasks	NUMBER: G3



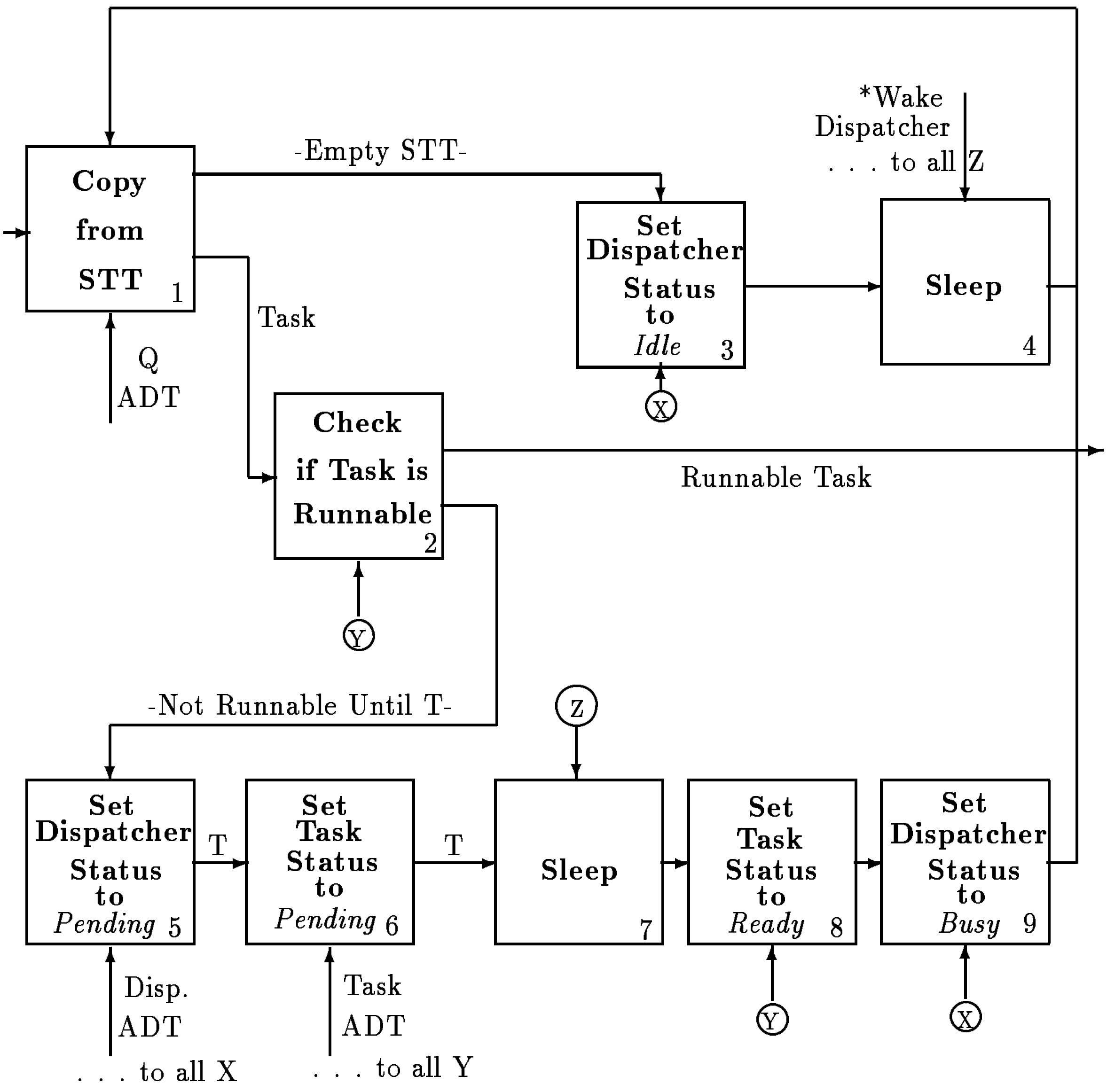
NODE: 32	TITLE: Local Scheduler - Heuristic	NUMBER: G6



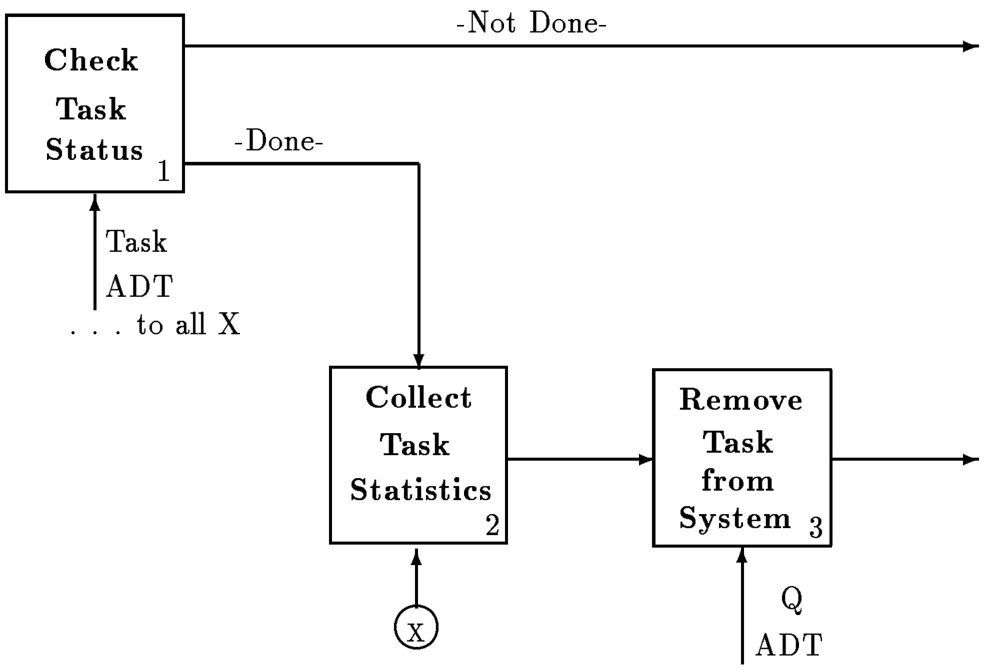
NODE:	TITLE:	NUMBER:
36	Dispatch and Execute Tasks	G4



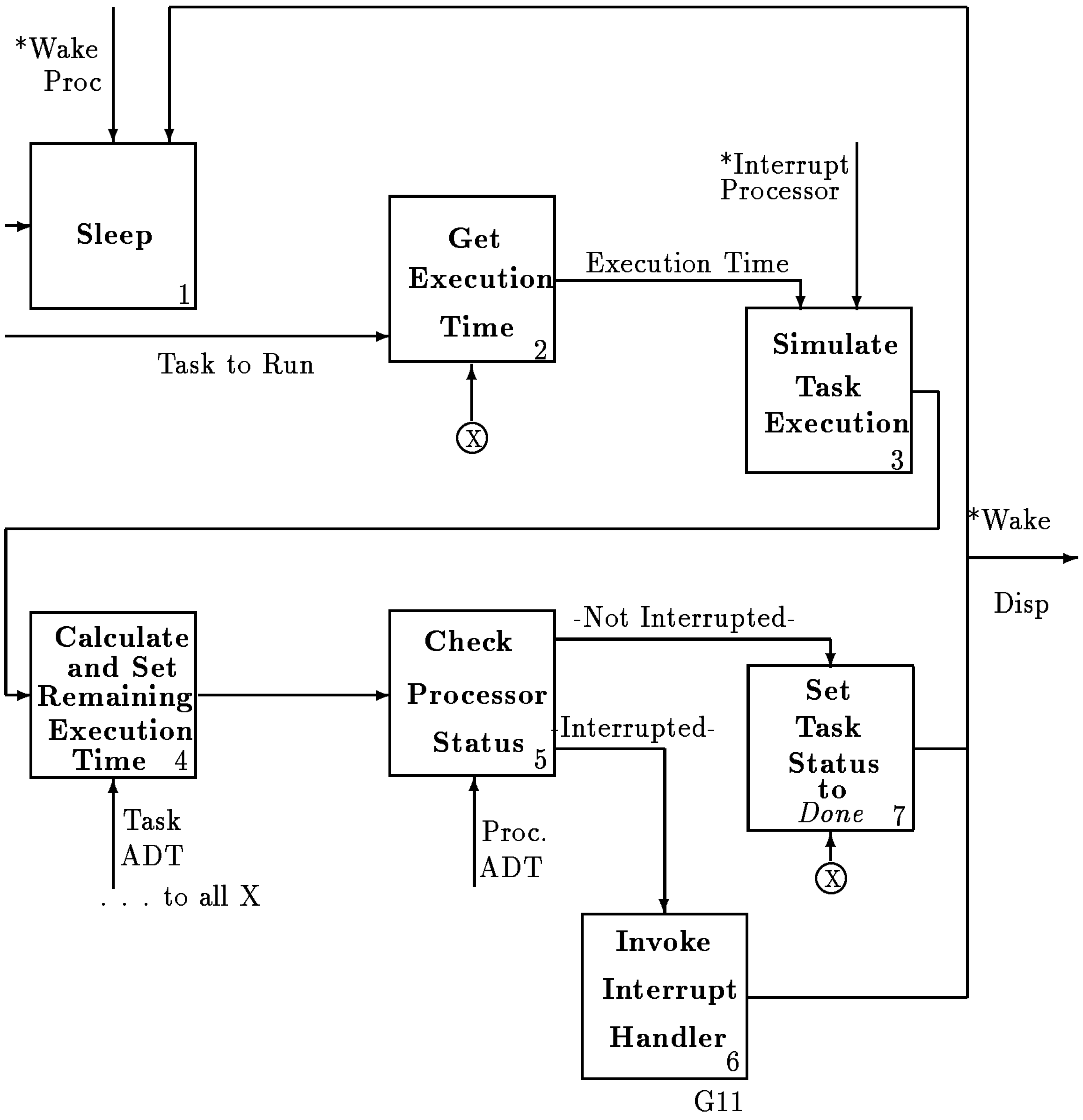
NODE: 361	TITLE: Dispatcher	NUMBER: G8



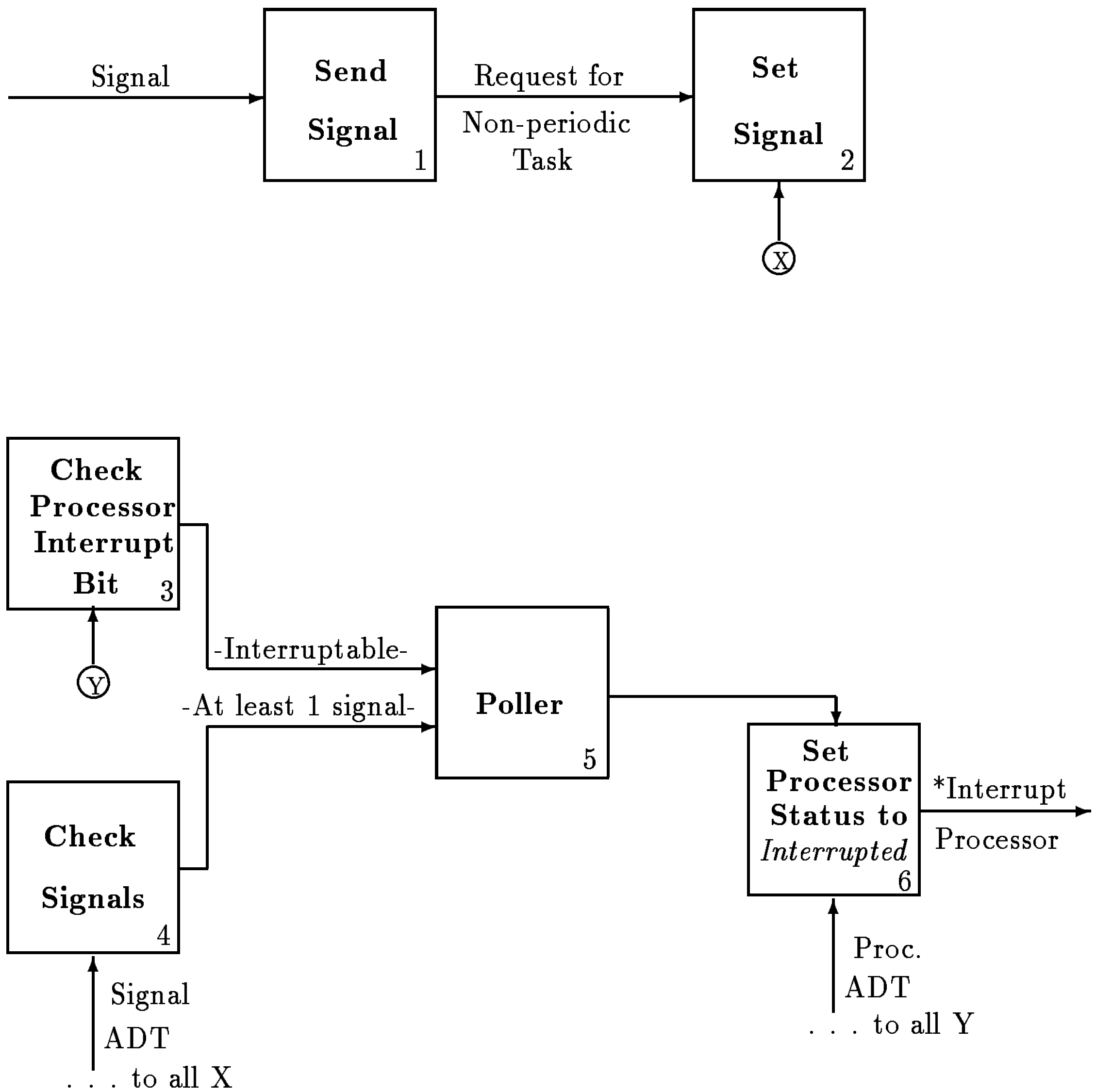
NODE:	TITLE:	NUMBER:
3611	Get Task to Dispatch	G12



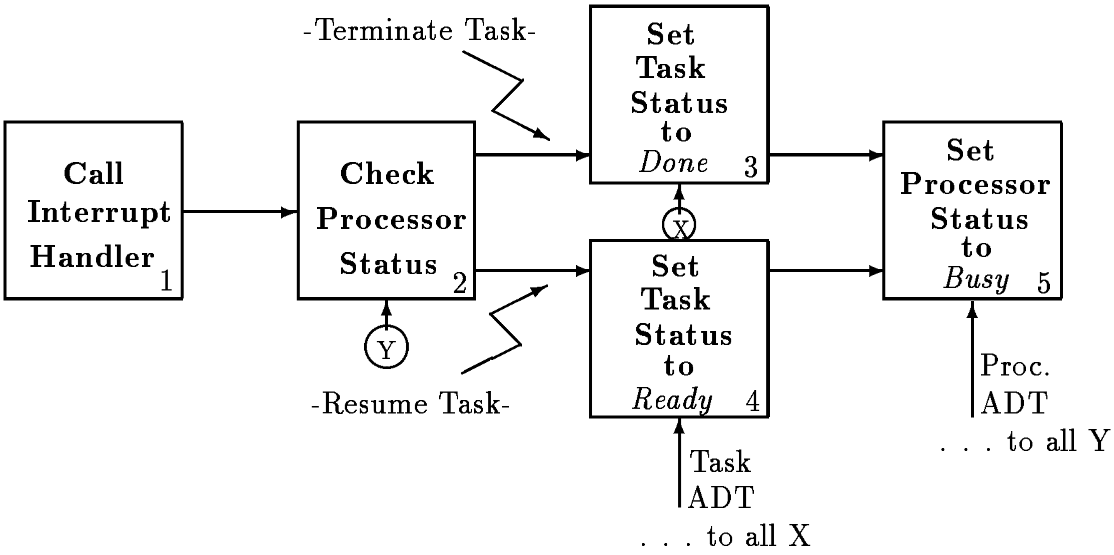
NODE:	TITLE:	NUMBER:
3619	Remove Dispatched Task	G13



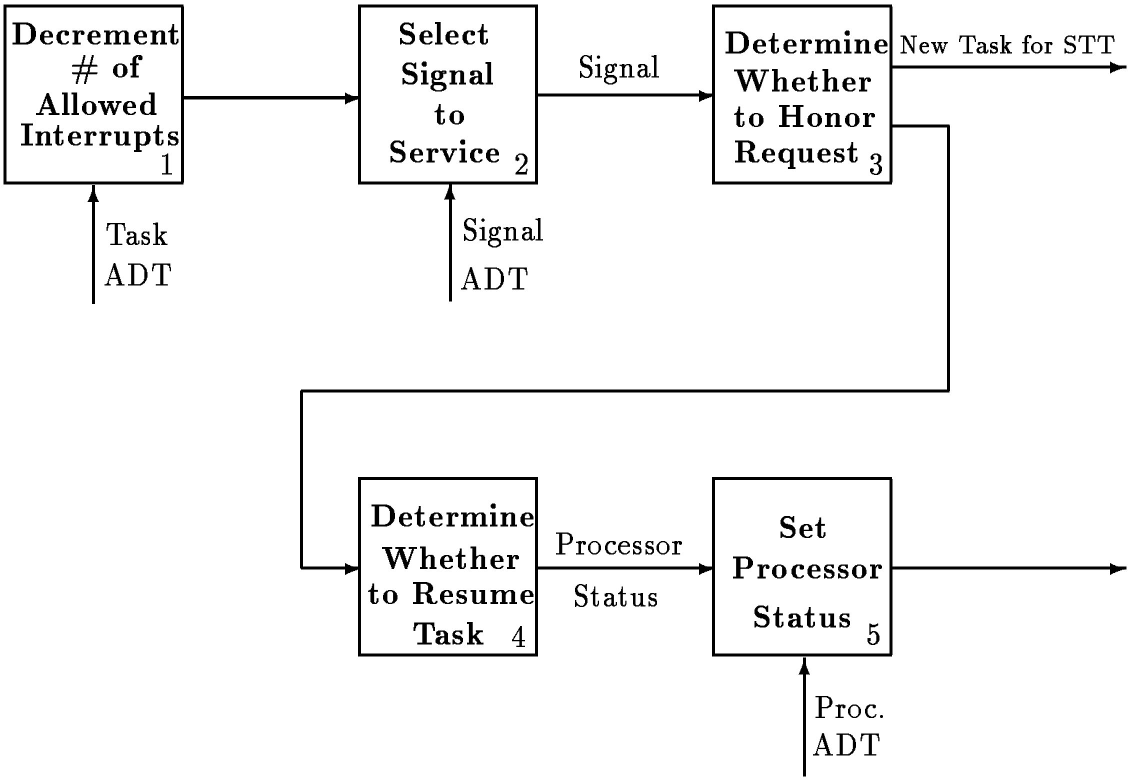
NODE: 362	TITLE: Processor	NUMBER: G9



NODE:	TITLE:	NUMBER:
314	Send/Service Signals	G5



NODE:	TITLE:	NUMBER:
3626	Invoke Interrupt Handler	G11



NODE:	TITLE:	NUMBER:
34	Interrupt Handler	G7

3 Low Level Design - Abstract Data Types

In this section I will present all my abstract data types. They are not complete specifications, but cover at least everything presented in this paper. There is at least a pair of routines for each element each the data structure, one to set the value, one to return the value. See the files **def.h, structs.h** and **(disp, func, node, proc, q, res, sig, task)-routines.c** for more detail.

3.1 Global Types

There are several types that must be global so many routines can use them:

type TaskStatus **is**
  NotReady, Ready, Executing, Pending, Done
type ProcessorStatus **is**
  Idle, Busy, Interrupted, IntResume, IntTerminate
type DispatcherStatus **is**
  Idle, Pending, Busy, StartExec
type SubnetStatus **is**
  Ready, Executing, Requested
type ResourceUsage **is**
  Unused, Shared, Exclusive
type BitVector **is**
  On, Off, Disabled
type SignalType **is**
  L. Sched, G. Sched, Bidder
type DispatchingOption **is**
  Bound, Independent
type TimeUnit **is**
  usec, 10 usec, 100 usec, msec, 10 msec, 100 msec, sec
type FunctionType **is**
  Reclaiming, L. scheduling, Guarantee, H-function,
  G. scheduling, Signal selection, Signal honoring,
  Task resumption, Polling, Interrupt handling
type CODEVECT **is**
  int (*[MAXCODE])();
type TIMEVECT **is record**

```
    V : int[MAXCODE+2];
end record;
type RESVECT is record
    V : int[MAXRESOURCE];
end record;
type SORTVECT is
    int [MAXSORTKEY];
```

3.2 Task

package Task **is**

    routine print_line;
    — part of statistics reporting
    routine stat_one_line;
    — part of statistics reporting
    routine stat_one_section;
    — part of statistics reporting
    routine task_(s,g)et_acomp;
    — set/get actual computation time
    routine task_(s,g)et_allint;
    — set/get number of allowed interrupts
    routine task_(s,g)et_arrival;
    — set/get arrival time
    routine task_(s,g)et_bcount;
    — set/get bid count
    routine task_(s,g)et_bidptr;
    — set/get bid pointer
    routine task_(s,g)et_bthresh;
    — set/get bid threshold
    routine task_(s,g)et_chandelay;
    — set/get channel delay
    routine task_(s,g)et_cnode;
    — set/get current node
    routine task_(s,g)et_code;
    — set/get code pointer
    routine task_(s,g)et_dest;
    — set/get destination node
    routine task_(s,g)et_dl;
    — set/get deadline
    routine task_(s,g)et_eat;
    — set/get resource vector
    routine task_(s,g)et_event;
    — set/get event number
    routine task_(s,g)et_finish;
    — set/get finish time
    routine task_(s,g)et_fna;
    — set/get 1st h-function value
    routine task_(s,g)et_fnb;
    — set/get 2nd h-function value

routine task_(s,g)et_fv;
— set/get h-function value
routine task_(s,g)et_h2nd;
— set/get 2nd lowest h-function value task
routine task_(s,g)et_hcount;
— set/get number of hops (travel between nodes)
routine task_(s,g)et_id;
— set/get id number
routine task_(s,g)et_issbid;
— set/get issued bids
routine task_(s,g)et_laxity;
— set/get laxity
routine task_(s,g)et_lst;
— set/get latest start time
routine task_(s,g)et_msize;
— set/get message size (when in transport)
routine task_(s,g)et_mtype;
— set/get message type (when in transport)
routine task_(s,g)et_name;
— set/get identifying name
routine task_(s,g)et_nbcount;
— set/get new bids count
routine task_(s,g)et_next;
— set/get next in queue (when in queue)
routine task_(s,g)et_nsurplus;
— set/get node surplus
routine task_(s,g)et_numcode;
— set/get number of code sections
routine task_(s,g)et_onode;
— set/get original node
routine task_(s,g)et_period;
— set/get period
routine task_(s,g)et_periodic;
— set/get periodic flag
routine task_(s,g)et_predcount;
— set/get predecessor count
routine task_(s,g)et_preempt;
— set/get preemptable flag
routine task_(s,g)et_prev;
— set/get previous in queue (when in queue)
routine task_(s,g)et_procid;

— set/get processor id
routine task_(s,g)et_rcomp;
— set/get remaining computation time
routine task_(s,g)et_readytime;
— set/get ready time
routine task_(s,g)et_resource;
— set/get resource usage vector
routine task_(s,g)et_rndtrpdelay;
— set/get round trip delay
routine task_(s,g)et_sentout;
— set/get sent out flag
routine task_(s,g)et_sft;
— set/get scheduled finish time
routine task_(s,g)et_size;
— set/get size
routine task_(s,g)et_sortkey;
— set/get sort key
routine task_(s,g)et_source;
— set/get source node
routine task_(s,g)et_sst;
— set/get scheduled start time
routine task_(s,g)et_start;
— set/get actual start time
routine task_(s,g)et_status;
— set/get status
routine task_(s,g)et_succlist;
— set/get successor list
routine task_(s,g)et_surplusw;
— set/get surplus
routine task_(s,g)et_taskptr;
— set/get task pointer
routine task_(s,g)et_timeout;
— set/get time out
routine task_(s,g)et_wcomp;
— set/get worst case computation time
routine task_alloc_more;
— allocate memory for subnet-related information
routine task_char_status;
— get readable status
routine task_collect_stats;
— tally statistics

routine task_copy;
— copy a task
routine task_copy_eat;
— copy the eat vectors
routine task_copy_resource;
— copy the resource vectors
routine task_create;
— create a task
routine task_destroy;
— destroy a task
routine task_exec_code;
— execute task code
routine task_fill_sysparam;
— initialize as a system task
routine task_incr_numrecv;
— increment number received at a node
routine task_init;
— initialize a task
routine task_init_eat;
— initialize the eat vectors
routine task_is_dummy;
— check if it is a dummy task
routine task_next_arrival;
— get the next non-periodic task to arrive
routine task_print;
— print task summary
routine task_print_identity;
— print shorter task identity
routine task_report_stats;
— top-level statistics routine
routine task_start_time;
— **resource reclaiming goes here**
routine valid_numskey;
— check validity of number of sort keys
routine valid_sortkey;
— check validity of sort key index
routine valid_task_codesec;
— check validity of code section index
routine valid_task_flag;
— check validity of task pointer flag
routine valid_task_status;

```
    — check validity of status
    routine valid_time_index;
    — check validity of index into time vector
private
    type TASKT is record
        Id : int;
        — id number
        EventNum : int;
        — event number
        Periodic : int;
        — 1=periodic, 0=nonperiodic
        Period : int;
        — period of a periodic task, 0 otherwise.
        Preempt : int;
        — 1=preemptable, 0=nonpreemptable
        ProcId : int;
        — the processor the task code resides on
        Status : int;
        — task status
        SortKey : SORTVECT;
        — all queues must be sorted by these keys
        NumAllInt : int;
        — number of allowed interrupts
        Name : char *;
        — name of this task
        ONode : int;
        — original node
        CNode : int;
        — current node
        Code : CODEVECT;
        — array of ptrs. to integer functions
        NumCode : int;
        — how many are to be used
        Arrival : int;
        — arrival time
        WComp : TIMEVECT;
        — worst case computation time
        AComp : TIMEVECT;
        — actual computation time
        RComp : TIMEVECT;
        — remaining computation time
```

DL : int;
— deadline
Laxity : int;
— run time laxity
LST : int;
— latest start time
SST : int;
— scheduled start time
SFT : int;
— scheduled finish time
Start : int;
— actual start time
Finish : int;
— actual finish time
Resources : RESVECT;
— resource requests
SortNext : TASKT *;
— next in SortKey order
SortPrev : TASKT *;
— previous in SortKey order
ProcNext : TASKT *;
— next for same processor
ProcPrev : TASKT *;
— previous for same processor
FV : int;
— scheduling function value, e.g. the H value
FnValueA : int;
— function value fields for scheduler's use
FnValueB : int;
— function value fields for scheduler's use
(Old,New)Eat(E,S) : RESVECT;
— temporary resource vectors used by guarantee routine
H2nd : TASKT *;
— for backtracking use
PredCount : int;
— number of immed. predecessors; default = 0
SuccList : TASKT *;
— list of immediate successors
ReadyTime : int;
— max. of the sft of its predecessors
Source : int;

```
        — task source node
        Dest : int;
        — remote node being sent to
        More : SUB_INFO;
        — pointer to block of info for subnet
        type SUB_INFO is record
            ChannelDelay : int;
            — channel delay
            RoundTripDelay : int;
            — round trip delay
            MsgType : int;
            — message type
            MsgSize : int;
            — size of short messages (RFB, INFO etc.)
            BidCount : int;
            — counts number of received RFB responses
            NewBidCount : int;
            — auxiliary counter
            BidThresh : int;
            — threshold for the number of received RFB
            TimeOut : int;
            — how long RFB responses will be waited for
            FirstBidPtr : TASKT *;
            — pointer to the first received bid response
            TaskPtr : TASKT *;
            — used only at RESPONSE.IN.Q
            IssuedBid : int;
            — number of expected responses
            SentOut : int;
            — sent out flag HopCount : int;
            — number of visited nodes
            Size : int;
            — size
            NodeSurplus : int;
            — sum of computations  (longest dl - current time)
            SurplusW : int;
            — surplus in portion of STT up to the current task
        end record;
    end record;
end Task
```

3.3 Queue

All task queues have the capability to be linked by both the task sort links and the task proc links. Generally, only the STT uses the latter.

package Q **is**
routine q_add_task;
— add a task to the queue
routine q_append_task;
— append a task to the queue
routine q_char_type;
— get readable queue type
routine q_copy;
— copy a queue
routine q_create;
— create a queue
routine q_decr_size;
— decrement size of queue
routine q_destroy;
— destroy a queue
routine q_empty;
— empty out a queue
routine q_fix_proc_ptrs;
— fix up the liked list by processor within queue
routine q_incr_size;
— increment size of queue
routine q_init;
— initialize queue
routine q_move_task;
— move task from one queue to another
routine q_prepend_task;
— prepend task to the queue
routine q_print;
— print all tasks in queue
routine q_remove_task;
— remove task from queue
routine q_(s,g)et_head;
— set/get first task on sort linked list
routine q_(s,g)et_id;
— set/get queue id

```
    routine q_(s,g)et_nodeid;
    — set/get queue node id
    routine q_(s,g)et_numskey;
    — set/getnumber of sort keys for this queue
    routine q_(s,g)et_size;
    — set/get size of queue
    routine q_(s,g)et_tail;
    — set/get last task on sort linked list
    routine q_(s,g)et_type;
    — set/get queue type
    routine q_swap;
    — swap tasks on two queues
    routine q_unlink_task;
    — unlink a task from a queue
    routine stt_add_task;
    — add a task to the STT
    routine valid_q_type;
    — check validity of queue type
private
    type QUEUET is record
        Head : TASKT *;
        — pointer to first task in sort list
        Tail : TASKT *;
        — pointer to last task in sort list
        Size : int;
        — number of tasks in whole queue
        NumSortKey : int;
        — number of sort keys in use for this q
        NodeId : int;
        — what node does this queue reside on
        Id : int;
        — id of this queue (referenced as (NodeId,Id)) pairs
        Type : int;
        — queue type
    end record;
end Q
```

3.4 Processor

```
package Proc is
    routine proc_char_status;
    — get readable processor status
    routine proc_create;
    — allocate memory for a processor record
    routine proc_destroy;
    — free memory used by processor record
    routine proc_init;
    — initialize processor record
    routine proc_print;
    — print processor record
    routine proc_(s,g)get_intbit;
    — set/get value of interrupt bit
    routine proc_(s,g)et_status;
    — set/get status
    routine proc_(s,g)et_task;
    — set/get pointer to current task
    routine valid_proc_status;
    — validate status
private
    type PROCT is record
        Status : int;
        — processor status
        IntBit : int;
        — processor interrupt bit
        Task : TASKT *;
        — task in execution
        DispRec : DISPT *;
        — the dispatcher record for this processor
    end record;
end Proc
```

3.5 Dispatcher

package Disp **is**
 routine disp_char_status;
 — get readable dispatcher status
 routine disp_create;
 — allocate memory for a dispatcher record
 routine disp_decr_qsize;
 — decrement the size of the dispatcher's queue
 routine disp_destroy;
 — free memory used by the record
 routine disp_incr_qsize;
 — increment the size of the dispatcher's queue
 routine disp_init;
 — initialize the dispatcher record
 routine disp_next_task;
 — get the next task to be dispatched
 routine disp_print;
 — print the dispatcher record
 routine disp_remove_task;
 — remove a task from the queue
 routine disp_(s,g)et_cutoff;
 — set/get the cutoff task pointer
 routine disp_(s,g)et_q;
 — set/get the queue pointer
 routine disp_(s,g)et_qhead;
 — set/get the queue head pointer
 routine disp_(s,g)et_qsize;
 — set/get the queue size
 routine disp_(s,g)et_qtail;
 — set/get the queue tail pointer
 routine disp_(s,g)et_status;
 — set/get the status
 routine valid_disp_status;
 — validate the status
private
 type DISPT **is record**
  Status : int;
  — dispatcher status
  Head : TASKT *;
  — pointer to first for this processor

```
        Tail : TASKT *;
        — pointer to last for this processor
        Size : int;
        — number of tasks in this list
        Cutoff : TASKT *[2];
        — pointer to last task for dispatch/first task to be rescheduled
        Q : QUEUET *;
        — the actual queue
    end record;
end Disp
```

3.6 Node

package Node **is**

routine node_(s,g)et_bthresh;
— set/get bids threshold
routine node_(s,g)et_chanspeed;
— set/get the channel speed
routine node_(s,g)et_dispopt;
— set/get the dispatching option
routine node_(s,g)et_disprec;
— set/get the dispatcher record
routine node_(s,g)et_gsp;
— set/get the global scheduling policy
routine node_(s,g)et_nproc;
— set/get the number of processor
routine node_(s,g)et_nstt;
— set/get number of STTs
routine node_(s,g)et_num;
— set/get the number of nodes in the system
routine node_(s,g)et_procrec;
— set/get the processor record
routine node_(s,g)et_q;
— set/get the specified queue header
routine node_(s,g)et_sigbit;
— set/get the signal bit
routine node_(s,g)et_sthresh;
— set/get the surplus threshold
routine node_(s,g)et_stt;
— set/get the STT queue header
routine node_(s,g)et_sttid;
— set/get the STT id
routine node_(s,g)et_surplus;
— set/get the surplus
routine node_(s,g)et_sysproc;
— set/get the system processor id
routine node_(s,g)et_timeout;
— set/get the timeout
routine node_init;
— initialize a node
routine valid_disp_option;
— check validity of dispatching option

```
    routine valid_node_id;
    — check validity of node id
    routine valid_stt_id;
    — check validity of STT id
private
    type NODET is record
        NumProc : int;
        — number of processors
        ProcRec : PROCT *;
        — pointer to processor record
        SysProc : int;
        — system processor id
        NumSTT : int;
        — number of STTs
        STTId : int[MAXPROC+1];
        — id of STT assigned to each processor
        DispOption : int[MAXPROC+1];
        — dispatching option - bound, independent
        STT : QUEUET *[MAXSTT+1];
        — array of pointers to STTs
        Queues : QUEUET *[MAXQ];
        — array of pointers to all queues that node has 1 of
        Signals : SIGT[SIG_NUM];
        — signal records
        ChannelSpeed : int;
        — for the subnet
        GSchedPolicy : int;
        — global scheduling policy
        Surplus : int;
        — node surplus
        SurplusThresh : int;
        — surplus threshold
        RFBTimeOut : int;
        — rfb time out
        BidThresh : int;
        — bids threshold
    end record;
end Node
```

3.7 Signal

package Sig **is**
    routine char_signal;
    — get readable signal name
    routine sig_handle;
    — this is the interrupt handler
    routine sig_pik_fcfs;
    — pick a signal by fcfs order
    routine sig_send;
    — send a signal
    routine sig_(s,g)et_proc_intr;
    — "fake" interrupt handler - just set proc. status to *int-resume*
    routine valid_signal;
    — validate signal type
private
    type SIGT **is record**
        Bit : int;
        — the actual bit value
        Time : int;
        — time it was last set
    end record;
end Sig

3.8 Function

This is where all the replaceable modules are managed. For more detail, see the Programmer's Guide.

package Func **is**
  routine func_cleanup;
  — free memory used by function tasks
  routine func_exec_one;
  — execute the currently selected function for the type
  routine func_get;
  — get a function task pointer
  routine func_init;
  — initialize the function structures
  routine func_init_one;
  — initialize a particular function type
  routine func_init_systask;
  — turn a function into a system task
  routine func_init_task;
  — turn a function into a task
  routine valid_func_type;
  — validate the function type
private
  FuncPtr : TASKT *[MAXNODE+1][NUM_FTYPE];
  — pointer to task structure of chosen function for each node for each type
  CurFunc : int [MAXNODE+1][NUM_FTYPE];
  — index of chosen function for each node for each type
  NumGen : int [NUM_FTYPE];
  — number of times the type has been executed - used for setting unique ids
  WComp : TIMEVECT [NUM_FTYPE][MAX_FUNC];
  — worst case compuation time
  AComp : TIMEVECT [NUM_FTYPE][MAX_FUNC];
  — actual compuation time
  Period : int [NUM_FTYPE][MAX_FUNC];
  — period
  Preempt : int [NUM_FTYPE][MAX_FUNC];
  — preemptable
  Resource : RESVECT [NUM_FTYPE][MAX_FUNC];
  — resource use vector
end Func