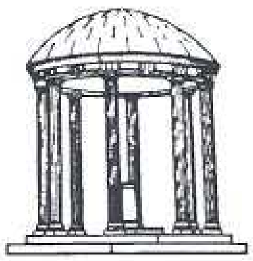
An N-body Simulation in a Virtual Universe Implementation Manual

TR92-046
Spring 1992

Adam Duggan
Erik Erikson
Carl Mueller
Mark Parris
Quan Zhou

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175



An N-body Simulation in a Virtual Universe

Implementation Manual

Project Members:
Adam Duggan
Erik Erikson
Carl Mueller
Mark Parris
Quan Zhou

COMP 145
John McHugh
Spring 1992
UNC Chapel Hill

An N-body Simulation in a Virtual Universe

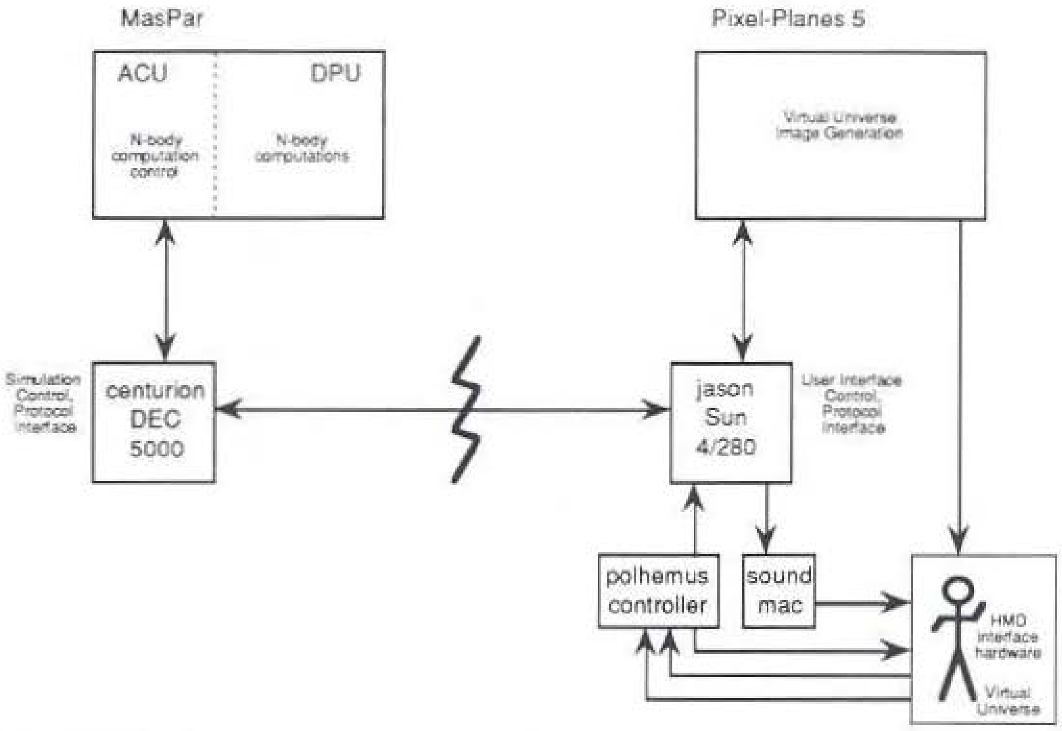
Implementation Manual

1. Overview

Carl Mueller

1.1 Introduction

The N-body Simulation in a Virtual Universe is a system of processes which computes and displays a set of interacting moving bodies in a three dimensional head-mounted display system. There are two distinct processes which are responsible for the interaction computation and user interface, respectively. These two processes may run on separate machines, and they communicate using a protocol library built for this purpose. The system is illustrated in the figure below. This manual will therefore describe each of these three major pieces in turn: the user interface, the protocol library, and the simulation computation.



1.2 High-level design issues

The statement of the project already dictated the division of the system into the separate simulation and user-interface processes. Breakdowns beyond this were up to the discretion of the team. Since it was deemed likely that several different simulation programs would be built, we were also given the requirement that the communications protocol be made explicit. However, we also recognized right away the need for a standard set of routines to use to actually implement the protocol. This is the purpose of the protocol library.

The HMD interface requires support for the various input and output devices associated with it. We are fortunate to already have access to several libraries of such routines created here at UNC. There are several lower-level libraries which are responsible for the gathering of tracker position and button-press data, the conversion of the position data into different coordinate systems, the output of predefined sounds, and the display of 3-D primitives in stereo. In addition, there is a library which handles the updating and control issues associated with virtual control panels containing buttons, dials, and sliders. These libraries proved essential in simplifying the design of the user-interface code.

The design of the communications protocol proved to be a catalyst for determining the exact nature of what may or may not be done by either the user interface or simulation. We wanted a great deal of flexibility in what sort of simulations could be implemented, while still having a user interface that made sense for each kind. In trying to determine the exact nature of the exchanges involved, we found we were either limiting one side too much, or else we were providing the opportunity for one side to confuse the other or the user. The road to the present setup was long and bumpy.

The nature of the simulation at first did not seem too important. We were provided with a simple, basic n-body simulation on the Maspar which seemed to provide reasonable results. We were also advised to use a solar system model as a simple test case for the simulation. We made one of the first test cases even simpler: one small body orbiting another large one. This immediately showed us that either: our simulator was much too inaccurate for the model, or else that the model was too precise for the given simulator. This issue was one which led us to the idea of supporting many different kinds of simulators, with the general n-body simulator being on one end and the orbital body ephemeris being on the other. Our support comes in the way of feature sets and having two distinct sets of bodies with different feature sets. This is discussed in the Protocol section.

The C programming language is used for all parts of the system with the exception of the calculation routines written for the Maspar. These routines are in mpl, the Maspar C-like parallel language.

1.2.1 Effects of Design on Testing

One of our goals from the start was to be able to test both the simulation and the interface separately. Because each is a separate program and all the communication is done through the protocol library, it was easy to create simple front or back-end code to test the simulation or user-interface, respectively.

1.2.2 Effects of Design on Assembly and Integration

The feature-oriented nature of the simulation means that if the user-interface doesn't yet have code implemented for a certain feature, we can simply declare that the simulation does not support that feature. This makes it easy to run without certain features and only have to worry about them when they become implemented, not before.

1.3 Common Data Structures

Several common data structures are used throughout the different pieces of the system. The purposes of these are described here. The complete definitions for the data structures may found in the include file 'nb_types.h', which, like all the others mentioned here, are found in the main include directory.

The data which describes a body is kept in one of the following structures:

```
typedef struct _body_type_ {
  XYZType position;
  XYZType velocity;
  NbColorType color;
  NbFloat charge;
  NbFloat mass;
  NbFloat radius;
  StatusType status;
} BodyType;

typedef struct _body_list_type_ {
  XYZType *position;
  XYZType *velocity;
  NbColorType *color;
  NbFloat *charge;
  NbFloat *mass;
  NbFloat *radius;
  StatusType *status;
} BodyListType;
```

These two types are necessary for performance reasons. The first structure is good for passing single bodies around. However, the second is better organized for moving around data for sets of bodies. This is because the individual fields are often shuffled around by themselves (position primarily), and thus 'irrelevent' data is not interleaved in between and moved around as well. In addition, this structure allows simulations that do not care about certain properties (such as color) avoid allocating space for them.

The status field above is used to inform the simulation if a body is new or disabled. In addition, the user interface uses this field to keep track of the selected, origin, and attached-to bodies. The macros CLEAR_STATUS, SET_STATUS, and CHECK_STATUS are provided for manipulating status.

The following structures keep track of information about the simulation and display settings:

```
typedef struct _law_type_ {
  char *law_name[LAW_NAME_SIZE];
  unsigned char num_parms;
  NbFloat parms[MAX_NUM_PARMS];
  NbFloat max_parms[MAX_NUM_PARMS];
  NbFloat min_parms[MAX_NUM_PARMS];
} LawType;


typedef struct _scenario_type_ {
  unsigned char num_laws;
  LawType force_law[NUM_LAWS];
  unsigned char selected_law;
  SampleType sample_ratio;
  TimeStepType time_step;
  int attached_to;
  int origin_body;
} ScenarioType;
```

The limits on the law name size and number of parameters are defined in the file 'nb_limits.h'. Information about ranges for variables other than law parameters are kept in the structure 'RangeType'.

The following structure is used to keep track of the features support by a simulation:

```
typedef struct _feature_info_type_ {
  short max_bodies;
  short min_bodies;
  short fixed_bodies;
  FeatureType feature_set[NUM_FEATURE_SETS];
  FeatureType world_features;
} FeatureInfoType;
```

The set of features is defined in 'nb_features.h'. Also included in this file are the macros CLEAR_FEATURE, SET_FEATURE, and CHECK_FEATURE for manipulating the different feature sets. A comprehensive description of the different features may be found in the protocol section of this document.

2. User Interface

2.1 Overview

Carl Mueller

2.1 .1 Introduction

The HMD interface immerses the user in its display, and thus regular keyboard interaction is no longer practical. It was therefore required that all interaction be controllable through the HMD interface. Given only a 3-D mouse and a low-resolution stereo display, the only practical control method that seems plausible is a system of virtual control panels, menus and tools. Several of the interface libraries are designed for supporting certain virtual tool operations such as 'grabbing' and 'flying.' Another more recent library supports control panels with buttons, sliders, and dials. From consideration of the support we had, of other programs' virtual interfaces, and of many hours of thinking, designing, testing, and redesigning, we came up with the interface presented in the User's Manual. Though there are still many optimizations that could be made to the interface, the design presented is relatively complete.

Using the libraries, many of the challenging implementation issues of the interface are solved. However, certain important issues remained. When wearing a HMD, it is desirable to keep the display updated continuously in response to user movements. Any sluggishness or glitches in the update tends to annoy the user or even cause nausea. The first issue of concern was therefore whether it was even possible to update a large number of moving bodies in real time. This is discussed below.

The necessity to do certain system I/O operations from the user interface (such as saving the current setup to disk) would prove to be a challenge since there is no guarantee of how long such operations might take. The present system in fact will pause and not update the display while such actions are occurring. Attempts were made to avoid this, however none proved entirely reliable.

2.1.2 Brief Description of Various Libraries Used

The libraries used by the interface are as follows:

- pphigs: responsible for display of graphic primitives on Pxpl5.
- trackerlib: takes care of finding position and orientation of tracking devices.
- adlib: keeps track of push-button devices (among others).
- vlib: deals with many common operations such as keeping track of different coordinate systems and implementing certain basic tools such as 'fly' and 'grab.'
- soundlib: handles the output of a set of predefined sounds.
- text: uses pphigs to display character strings on Pxpl5.
- ETK: handles interactions dealing with control panels.

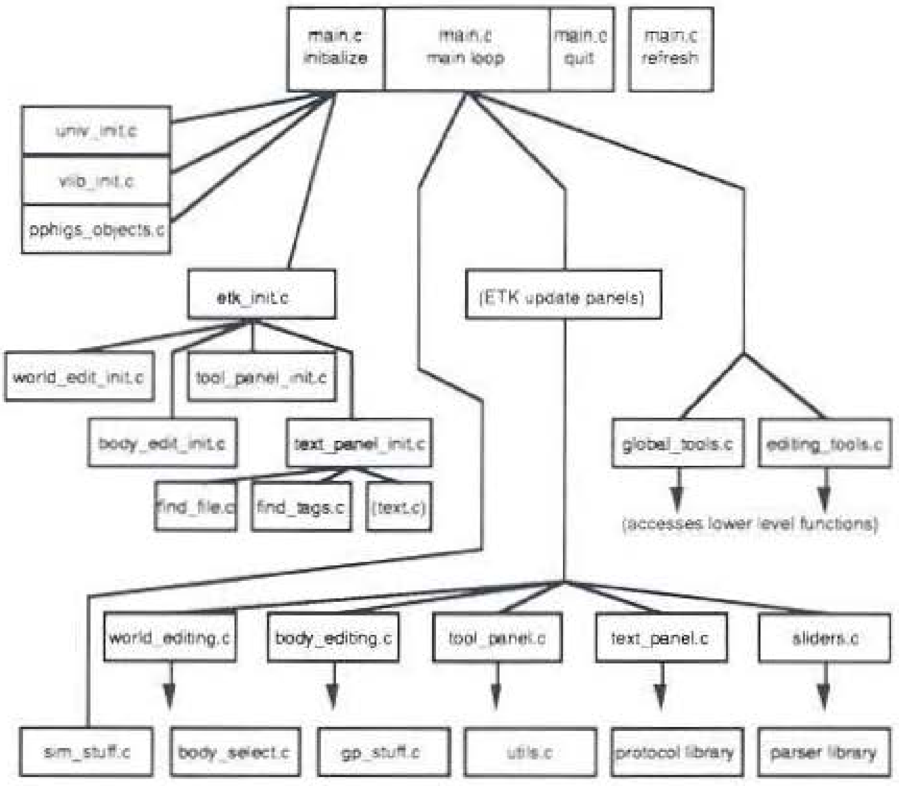
2.1.3 Description of ETK Toolkit Routines

Since the ETK library is responsible for a good deal of the implementation, we give some special attention to its functionality. Using this library, one can define control panels which may contain a set of buttons, dials, sliders, and displays. The tools are described to the library by pointers to their graphic structures, descriptions of their sizes and placements, and routines to call when they are activated. ETK is then responsible for the proper display and manipulations of the tools themselves, operations dealing with the control panels as a unit (for example, moving the entire control panel around), calculating the scaled values for the tools, and calling the call-back routines when necessary. The call-back may then take care of any additional graphic or data updates associated with the operation of that tool. Thus, for the most part, the description of the user-interface implementation boils down to describing the layout of the tools and the ETK call-backs.

2.2 Layout of the Interface Code

Carl Mueller

As described in the User's Manual, the interface paradigm revolves around the use of control panels and tools. Thus most of the user interface code is concerned with the setup and management of control panels and their associated widgets, and with the handling of tools. Underlying these routines are a variety of support functions in various files. The code files are organized in the following manner:



2.3 Data Structures and State Information

Carl Mueller

The design of the user interface data structures would prove to be a large and complex issue. The ETK call-back routines are passed a single user data variable, and yet the routines must be able to change all of the user interface state. There are several ways to handle this problem.

First, one could simply declare all the state information in global structures. However, this solution violates the principles of structured programming and can easily lead to messy code. We chose to avoid this route as much as possible.

Another solution would be to put all of the common state information in a single structure, a pointer to which is then passed to the call-backs. While this solution can also lead to some of the pitfalls of global variables, careful programming practices can avoid most of them. If the data structures within the structure are kept organized and common routines are used to access them, then this scheme will prove serviceable. In fact this is the organization we chose. The remaining drawback of this choice is that each time the large structure is changed, nearly everything must be recompiled.

The third solution avoids this problem and, in addition, is the method of choice for future implementations of this system. Instead of defining all the sub-structures within the main structure, only define pointers. For the sake of prototype checking, these pointers should all be to unique types, however the types they point to are not fully described by any global include file. Rather, the various sub-structures are defined in separate files which are only included by the routines that use them. The pointers in the overall structure must then be cast to the correct type that they really point to.

Here is a summary of what is contained in the large structure named 'UniverseType':

- scenario, range, and feature data
- body data and the default body description
- information necessary to properly display the bodies:
  what is the current transformation on the bodies?
  are vectors being shown?
  are disabled bodies being shown?
- whether or not the simulation is currently alive
- the current major mode (world edit/body edit/run/stopping)
- detailed control panel information:
  what are the current and previous panels?
  pointers to the panel's ETK structure, as well as pointers for all its widgets
- tool information:

what are the current and previous tools?

what are the tool pphigs names and important pphigs addresses?

- text panel information:

  what set of strings are being displayed?

  what are the pphigs addresses for these strings?

- numeric slider information:

  what variable is being modified?

  what are the slider ranges for the variables?

  what are the bounds for the variables?

- other misc information:

  which user? has he quit?

This is the 'global' state information, and it is defined in the include file 'nb_ui_types.h'. Most other state information is kept in the tool or button call-back routines. For instance, the fly tool routine itself keeps track of whether or not it is currently engaged.

There are a small number of global variables that are not included in the universe structure. These include the vlib-defined globals and the strings which contain the entry, scenario, and configuration file names. In the future some of these should be incorporated into the universe structure.

2.4 User Interface Code Specifics

2.4.1 Initialization

Carl Mueller

The file main.c includes the steps for initialization as well as the main loop. We now describe the sequence of events that occur during initialization time:

- parse command-line options
- check existence of data files
- initialize various fields in the universe data structure
- initialize vlib, tracker, sound, etc.
- read in pphigs data (fonts, etc.)
- initialize all control panels and other graphic objects
- initialize the GP call-back routine*
- initialize the simulation
- enter the main loop

* This routine handles the rapid display of the bodies and is discussed in a later section.

2.4.2 Main Loop

The main control loop includes the following steps:

- check if a simulation message has arrived
- process input for each user: *

  read the tracker and update the user transform matrices

  read the push-buttons

  update the users with respect to moving bodies (if necessary)

  call the ETK control panel routine for the active panel

  if no panel action happened, call the active tool function

  if no tool action happened, see if the tool panel was toggled
- update the GP call-back
- see if there's any terminal input
- update the user's display

* At the moment, support is only provided for a single user. A fair number of changes would actually be necessary to support more than one user, despite some of the support already included.

2.4.3 ETK-related Routines

Most all of the functions provided by the user interface are implemented either through ETK call-backs or through tool calls. The selection of tools in fact usually occurs as a result of a button call-back.

Definitions of the ETK control panels and their widgets occur in four different files:

- world_edit_init.c
- body_edit_init.c
- tool_panel_init.c
- text_panel_init.c

Each setup routine proceeds as follows: all the ETK widgets are declared, they are then grouped into their associated panel, and finally the actual placements of the widgets are determined.

In addition, several header files are involved:

- etk_init.h
- world_edit.h, body_edit.h, tool_panel.h, text_panel.h

The former includes enumerated type definitions which allow giving all the buttons names. The latter files include the function prototypes for all the call-backs.

The call-backs themselves are defined in one of the following files:

- world_editing.c
- body_editing.c
- tool_panel.c
- text_panel.c
- sliders.c

This organization works well for widgets that are found only on one single panel. However, some of the widgets are located on both the body and world editing panels. The slider related call-backs are all found in sliders.c. For buttons such as 'go to body', the call-backs were placed in world_editing.c.

If one wanted to add another button to one of the existing panels, the following steps would be required:

1. Create a pphigs icon for the button and make sure it gets loaded by one of the routines in pphigs_objects.c.
2. In etk_init.h update the number of buttons and their enumeration for the appropriate panel.
3. Add the ETK call to create the button in the appropriate init file, as well as the code to position the button properly.
3. Write the call-back routine itself and place it in the appropriate file.

2.4.4 Tool Routines

The tool functions are what control the actions of the various tools the user may use, including the global fly, grab, scale, and select tools, as well as the more specific add, position, radius, and velocity tools.

Definitions of these routines are found in one of the two files:

- global_tools.c (with header file global_tools.h)
- editing_tools.c (with header file editing_tools.h)

The tool pphigs structure names are kept track of in the tool information in the universe structure. This array is actually set up in pphigs_objects.c. Also kept in the tool information structure are pphigs pointers to display list elements that must be updated when using a particular tool. Finally, the tools are enumerated in nb_ui_types.h.

2.4.5 Miscellaneous Routines

There are many files containing various support and service routines. These are described here.

The file gp_stuff.c contains, in addition to routines which talk to the GP call-back (see below), routines which are used whenever a body parameter must be modified. Other routines may look at the values in the body list, but should not modify them directly, or else the bodies may not be displayed properly.

The file sim_stuff.c contains for the most part routines to talk to the simulation. In addition there are routines to properly set up the control panels whenever a new scenario or simulation is loaded.

The file body_select.c contains a package of routines for dealing with the currently selected body. In addition there is a routine named select_update_panels which updates the state of the control panel depending upon the state of the selected body.

Some important routines, which are currently located in inappropriate files for the most part, are:

- Switch_To_Control_Panel: remembers the last panel, toggles up the given panel.
- Switch_To_Tool: remembers the last tool and switches to the given one.
- Switch_To_Strings: switches the text panel display to show the given set of strings.

There are several files containing other various support routines:

- find_file.c:
  get_file_list: gets a list of files with a given extension that are found in a given directory.
- find_tags.c:
  get_tag_list: parse the configuration file to find out the names of the simulation tags.
- support.c:
  Parse_Command_Line: interpret command line options.
  Usage: print out a message displaying proper command usage.
  Handle_Terminal_Input: handle characters from the keyboard.
- utils.c:
  RGB_to_HSL, HSL_to_RGB: convert a color from one system to the other.
  fix_bounds_float, fix_bounds_char: make sure a value is within limits.
  basename, dirname: parse elements from a file pathname.
  lookup_string: finds where a string is in a list of strings.

The remaining files which have not been mentioned yet contain initialization routines:

- univ_init.c
- vlib_init.c
- etk_init.c

2.4.6 Description of the GP call-back *Carl Mueller*

As mentioned before, it is imperative to update the display of the moving bodies in real time. The pphigs interface to Pxpl5 is not very appropriate for changing the positions of many independent bodies at once. This is because of the communications bottleneck between the Pxpl5 host and the graphics processors (GPs). We found that by using regular pphigs updates frame rate slowed down appreciably when changing as few as one hundred bodies. We therefore had to find a better way around this update problem. The solution proved to be what is referred to as the GP call-back.

The update problem has to do with the amount of data that is sent from the from host to the GPs. Under pphigs, a single sphere update sends a relatively large sphere message which includes a wide set of information about the sphere. However, during run mode, the only value being changed is the sphere's position. We can send just the position information to the GPs. This is done using ROS routines, the Ring Operating System which pphigs itself is written over. Then, by inserting a call-back structure into the pphigs display-list, the sphere information can be read by the call-back code and used to directly call the GP low-level draw sphere routine. This method allows several hundred spheres to be updated without appreciable loss of frame rate.

The GP call-back is independent of the rest of the user-interface code. That is, it is compiled separately to run on the Pxpl5 GPs, not on the front end. It is possible for this code to establish its own data structures and event set up event-handler routines such that it can respond immediately to messages sent to it. We have decided then to implement an interface with the GP call-back to handle the following actions:

- send a new set of body positions to GP
- send a new set of velocities to GP
- send a new set of radii to GP
- send a new set of colors to GP
- send a new set of statuses to GP
- change status of displaying vectors or disabled bodies

To speed up as much as possible the front end side, we chose to have the host send the new information in the form of a single message to the first GP. This GP then distributes the information out to the other GPs in the ring. However, this also limits the number of bodies since the GP input FIFO is of limited size. If the current limit of 500 bodies ever proves too small, then this interface will have to change somewhat. The change would require having the host (instead of the first GP) distribute the messages.

The design of the GP call-back proved to be challenging. The basic problem is the uncertainty of the timing between updating the body information and displaying it. When a Pxpl5 draw frame command is issued, it will actually be some time before the GP call-back code is actually called. In the mean time, we might already have a new set of body data to send down. The solution is to use a queue and a set of semaphore variables between the message handlers and the drawing code.

All the GP call-back-related front-end code is self-contained in the gp_stuff file. This will enable easier portability should to opportunity arise to switch to a different graphic architecture.

2.4.7 Maintaining Continuous Updates *Carl Mueller*

Since some of the functions involve system I/O operations with no guaranteed response time, some thought has been put into how display updates could be maintained. The method chosen involves the use of the Unix interval timer interrupt routine to generate a SIGALRM. This signal is then caught and used to call a special display update routine. This handler will do a subset of the events handled by the main loop: those directly related to updating the display. Since no user interaction is possible during this time, the cursor will be changed to a traditional clock-like icon indicating that the user must wait.

To avoid conflict with the main line routines, this handler is only enabled when possibly long operations may occur. The particular operations where this handler is necessary are the load scenario file, save scenario file, and load simulation operations.

These handlers are defined at the end of the main.c file:

- refresh_init: sets up the refresh interrupt handler.
- refresh_on: turns on the refresh system.
- refresh_off: turns off the refresh system.
- refresh_int: the actual interrupt handler.

Unfortunately, these routines proved to be somewhat unstable, sometimes resulting in program lock-up. The problem seems to involve collisions of interrupts. Pxpl5 generates an interrupt whenever a message is sent to the host machine. While this message is being read, no other interrupts should occur. We attempted to set up the clock timer to avoid such collisions, and yet they still managed to occur.

Thus by default the refresh system is disabled at startup. If one wishes to experiment with the refresh system, then one may give the '-r' option on the command line to enable it.

2.4.8 Scenario File Implementation *Adam Duggan*

The scenario file for the nbody application is an ASCII text file defining the complete state of the simulation at the point the user saved it (see the nbody user manual for a complete description). Its ASCII format (as opposed to binary) makes it simple for the user to edit the file, or even write his own from scratch. Due to the relatively loose formatting requirements of the file, a Lex and Yacc parser system was chosen to read it in. A conventional C parser would require a fair amount of logic just to handle varying white space, whereas the Lex parser takes care of this automatically. Also, changing the specifications of the file are much simpler when implemented this way versus manually. For documentation on the use of Lex and Yacc, see the Unix Programmer's Manual, Supplementary Documents I (chapters 15 and 16).

Lex and Yacc are tools specifically designed to aid programmers in the generation of input parsers. Lex is a lexical analyzer which allows one to define regular expressions describing pieces of input data. It translates each piece into a token which is then processed by Yacc, the semantic parser. Yacc uses a context free grammar along with a set of actions (C source code) to process the input. Both the Lex source file and the Yacc source file are run through their respective compilers to generate C source files.

Since most of the data from the scenario file is of the form *keyword numerical_value* , most of the rules took this same form. The actions taken on each of these rule inputs are defined such that all of the input data is conditionally placed in the proper data structures, as the simulation may not allow certain fields to be changed. Additionally, the rules had to be defined such that they would allow only specific keywords and data within a *body* declaration. This data may come in any order, and there is not a set amount of it. Yacc's recursive rule definitions make handling variable amounts of data simple. A simple rule takes care of checking most errors as the file is read.

2.5 Changes for the Future

Carl Mueller

There are a few major changes that we wish to implement in future versions of the code:

- Change the universe data structure as mentioned in the state section.
- Move the functions around to different files:
  All functions which update panels or switch tools should be together
  Perhaps there should be a file of ETK call-backs which are common to several modes.
- There are several operations that are defined repeatedly. These should be consolidated into appropriate functions. An examples is deriving the average eye transformation.
- The placement and sizing of control panel panel widgets perhaps can be made easier to modify.

2.6 Testing

Due to the large number of functions, tools, and features available in this system, there proved to be no very straightforward method of testing. Also, being immersed in the HMD means that one cannot be interacting with the console at the same time. As a result of these factors, one must enter the system and very carefully test out each button, slider, dial, display, and tool under various conditions. Doing this properly also means having a variety of simulations available which allow different sets of features. This process can be rather time-consuming. It is best to have another person nearby to make a note of any deficiencies found since making such notes while wearing a head-mounted display can be awkward.

3. Protocol Implementation

Mark Parris

3.1. Purpose

The project specification for the Nbody Simulation in a Virtual Universe explicitly states that the simulation and the user interface run as separate processes and communicate over a well documented protocol. The protocol allows the user the flexibility of writing simulations besides the one provided with this package. Using TCP/IP on a local area ethernet allows medium bandwidth communication with reliable delivery. Since TCP/IP provides the performance needed by this application, protocols requiring more complex semantics, such as UDP's unreliable and out of order delivery, can be avoided. Other alternatives such as pipes or a different type of network were not considered because of their unavailability in the development environment.

The protocol is implemented as a library (libnbp.a) available on the Sun (SunOs), DEC (Ultrix), and Vax (Ultrix) platforms. Looking up a process entry in a process configuration file (*see Process Configuration Specification in the User's Manual Appendix*), starting up a remote process, determining the set of features supported by the simulation, converting data between the architectures of different machines, and exchanging configuration and body information are all supported by this library. All of these features are described in more detail below.

The protocol specification includes a description of features, a method for allowing the simulation to specify what features it supports, a description of the message sequences, a description of the expected data conversion, and a description of the message formats.

3.2. Features

Supporting simulations with different sets of features was a primary goal of our design. We wanted to support simulations where the bodies move in orbits and the user can't affect anything but color and radius of the bodies. We also wanted to able to support n-body simulations where the user may alter the velocities, masses, charges, positions, and radii of bodies as well as add and delete them. Further, we wanted to support simulations that had both types of bodies, with the masses of the orbital bodies affecting the n-bodies but not the reverse, and with the user interface able to change different sets of properties of the bodies of each set. To do this we decided to provide a wide range of tools and operations on bodies, force laws, and the properties of each and to allow the simulation to specify which of them it supports. Further, to support the third case above, we decided that we needed multiple body sets: alpha bodies and beta bodies. This specification occurs by an exchange of features in the Init Mode of the protocol. Once these features are exchanged the user interface disables tools that the simulation does not support and runs as usual.

3.2.1. Feature Sets

As indicated above two sets of bodies with different features are allowed. The simulation may specify an alpha set of bodies, from which bodies may be neither added nor deleted. This set is intended primarily to contain those bodies where there is a fixed equation associated with each body. Since this one to one correspondence exists, adding bodies hardly seems practical, and deleting serves little purpose other than removing that equation from display forever. However, one can enable and disable these bodies as described below. The beta set of bodies. may have bodies added and deleted (if the feature is set). Both sets of bodies can support any of the features and each set has its own feature list. At configuration time, the simulation tells the user interface how many bodies are in each set.

Here the features are broken down into groups with an overview of the features in that group followed by notes about individual features as appropriate. Note that turning on and off features is supported through the use of macros in nb_features.h.

> `SET_FEATURE(alpha_features, CHANGE_MASS, NB_ON)` turns the CHANGE_MASS feature on.
>
> `CLEAR_FEATURE(alpha_features)` clears the entire set of features for alpha_features.
>
> `SET_FEATURE(alpha_features,CHANGE_MASS,NB_OFF)` turns off the mass feature.

The feature flags are discussed below. The flag name used with the macros is specified in parentheses (e.g. changing mass (CHANGE_MASS)).

3.2.2. Turning Bodies On and Off

The user interface supports two ways of choosing to include bodies in a particular simulation. Adding (ADD_BODY) and deleting (DELETE_BODY) actually change the list of bodies in the simulation, increasing or decreasing the number of bodies sent between the simulation and the user interface. However, there may be models where a deleting a body (and thus rearranging the list) may have undefined consequences, such as deleting the fifth body in a solar system simulation. Do all of the other bodies move up, with Saturn taking Jupiters orbit? To allow turning off a body without rearranging, a list disable (DISABLE_BODY) is provided. Disabled bodies remain in all lists of bodies but they are not displayed. The simulation is free to interpret this disabling as it sees fit (provided that it keeps the body in its lists) or to ignore the feature altogether.

Note: ADD_BODY and DELETE_BODY are ignored when they occur in the feature list for the fixed body set.

3.2.3. Changing Body Parameters

A simulation may choose to allow the user interface to change any subset of the body parameters. If a simulation doesn't allow the user interface to change a particular body parameter it must provide the values itself if they are of any concern. No modification of these fields, either by editing, loading, or defaults may occur as a result of the user interface if the simulation hasn't enabled the feature. The features are listed below. Turning one of them on allows the user interface to support the action.

CHANGE_MASS
CHANGE_POSITION
CHANGE_VELOCITY
CHANGE_CHARGE
CHANGE_COLOR
CHANGE_RADIUS

3.2.4. Caring About Body Parameters

In addition to permitting change to body parameters, there is a question of concern for the values of parameters. Two of the body parameters merit special consideration. Color and radius are visible properties of the bodies which must exist. If the simulation disallows changing them, it must provide default values. Further, color and radius are usually not of concern to force law calculations or simulations but there are cases, such as collisions, or a simulation of sorting based on color where the simulation may care about these properties. To avoid unnecessary overhead, color and radius are not usually sent from the user interface to the simulation. However, by setting the features SEND_COLOR and/or SEND_RADIUS the simulation can indicate that the user interface should send these properties as well.

3.2.5. Changing Scenario Information

A simulation may choose to allow the modification of the time step (CHANGE_TIMESTEP) or sample ratio (CHANGE_SAMPLE_RATIO) used to calculate and control the display rate of body positions. Further it may choose to support multiple force laws as well as adjustable force law parameters. Note that there are no features associated with this since the existence of multiple laws implies the permission to choose among them and the existance of a coefficient implies that it is adjustable.

3.2.6. Interactive Parameters

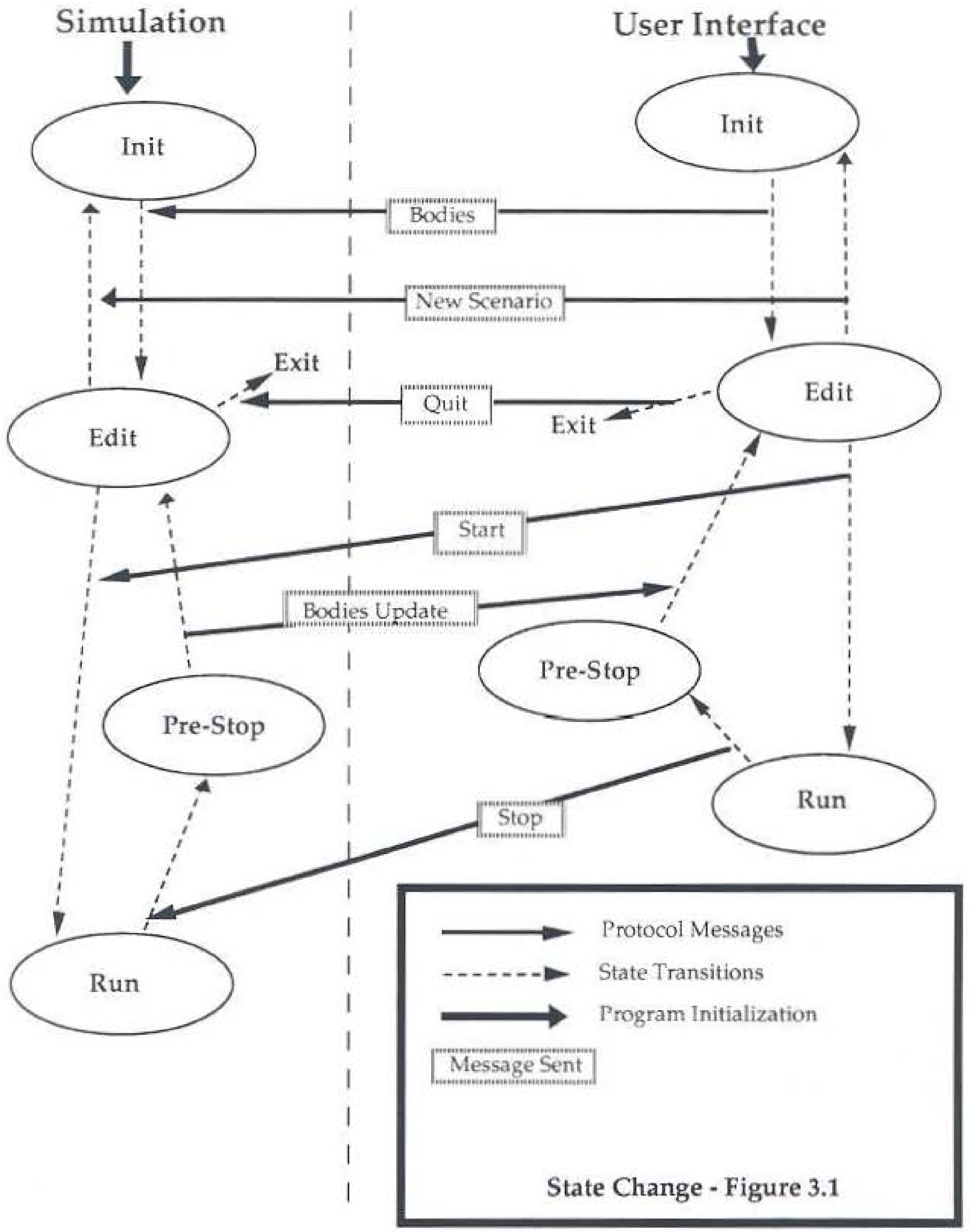
In addition to the changes supported during editing mode, the user interface supports two actions during run mode. The user interface may send thrust updates or sample ratio changes up to once per frame of data. As usual, the simulation can choose to support these tools by enabling CHANGE_THRUST and/or CHANGE_INTERACTIVE_SAMPLE_RATIO.

3.2.7. Miscellaneous

Finally, the user interface provides the capability of displaying velocity vectors during editing sessions and during run mode. However, this features is of no use if the the simulation has no concept of velocity. For instance, an orbital simulator may calculate the positions of the planets at time increments but have no need to calculate the velocity of the planets at any point in time. In such a case the simulator would not enable the SHOW_VELOCITY feature.

3.3. Message Sequence

This section defines the expected sequence of messages and the different states each side of the protocol can be in. Figure 3.1 shows the possible states and the transitions that may occur between them. Both processes start in Init mode, passing quickly to Edit mode. Once past the initial Init mode both processes cycle from Edit mode to Run Mode to Pre-Stop mode and back to Edit mode depending on the user's action and resulting messages sent. It is also possible to return to Init Mode or to exit from Edit Mode. The implementation of either process may of course use other internal modes but the ones listed above are all that the simulation need be aware of.



State Change - Figure 3.1

3.3.1. Init Mode

As shown in Figures 3.2a and b, the protocol proper begins with the simulation sending a CONFIG message to the user interface. It should be noted that the protocol library begins one step before this with the user interface starting the simulation as a remote process using the *nbp_run* function. Starting the simulation from the user interface is not a formal part of the protocol, but rather an additional feature of the library. The configuration message specifies a great deal of information about the simulation. Included in this message may be the number of bodies in the fixed set of bodies, the maximum total number of bodies, the minimum number of enabled messages, the features of the sets of bodies (both sets if both can exist), default color and/or radius for the beta set of bodies if the user interface can't change color and/or radius, ranges on the properties of each set, the number of laws, their names, and default parameters and ranges. Additionally, defaults and ranges for the time step and sample ratio may be included. For a description of the conditional creation of the CONFIG message see the *message format* section below.

After sending the initial configuration message the simulation may conditionally send a BODIES message describing those bodies which cannot be added or deleted. If all bodies can be added or deleted the message contains zero bodies. If the total set of bodies includes a fixed set of bodies, then at least those bodies are described via this message. The number of bodies in this case would be the number of bodies in the fixed set. In addition, if the simulation doesn't allow adding bodies in the second set those bodies are included in the BODIES message as well. In this case, the number of bodies in the message will be the maximum number of bodies. The properties which may be conditionally included in the BODIES message are declared in the *message format* section below.

Following the transmission of the body message, the user interface should load in whatever information it may wish to about bodies and the scenario state. Following this, the user interface should send the simulation an initial SCENARIO message indicating the currently selected law, the number of parameters sent, and, where applicable, the time step, sample ratio, and force law parameters. For more information on the conditions under which specific items of data are sent, see the *message format* section below.

Next, the user interface should send a BODIES message to the simulation, with the newly added bodies marked as NEW in the status field. The simulation is expected to receive this message and fill in any parameters that the user interface isn't allowed to change and send a BODIES message back to the simulation so that the simulation has a fully initialized set of bodies for display.

The transmission of this message on the simulation side, and the receipt of the message on the user interface side causes a transition to Edit Mode.

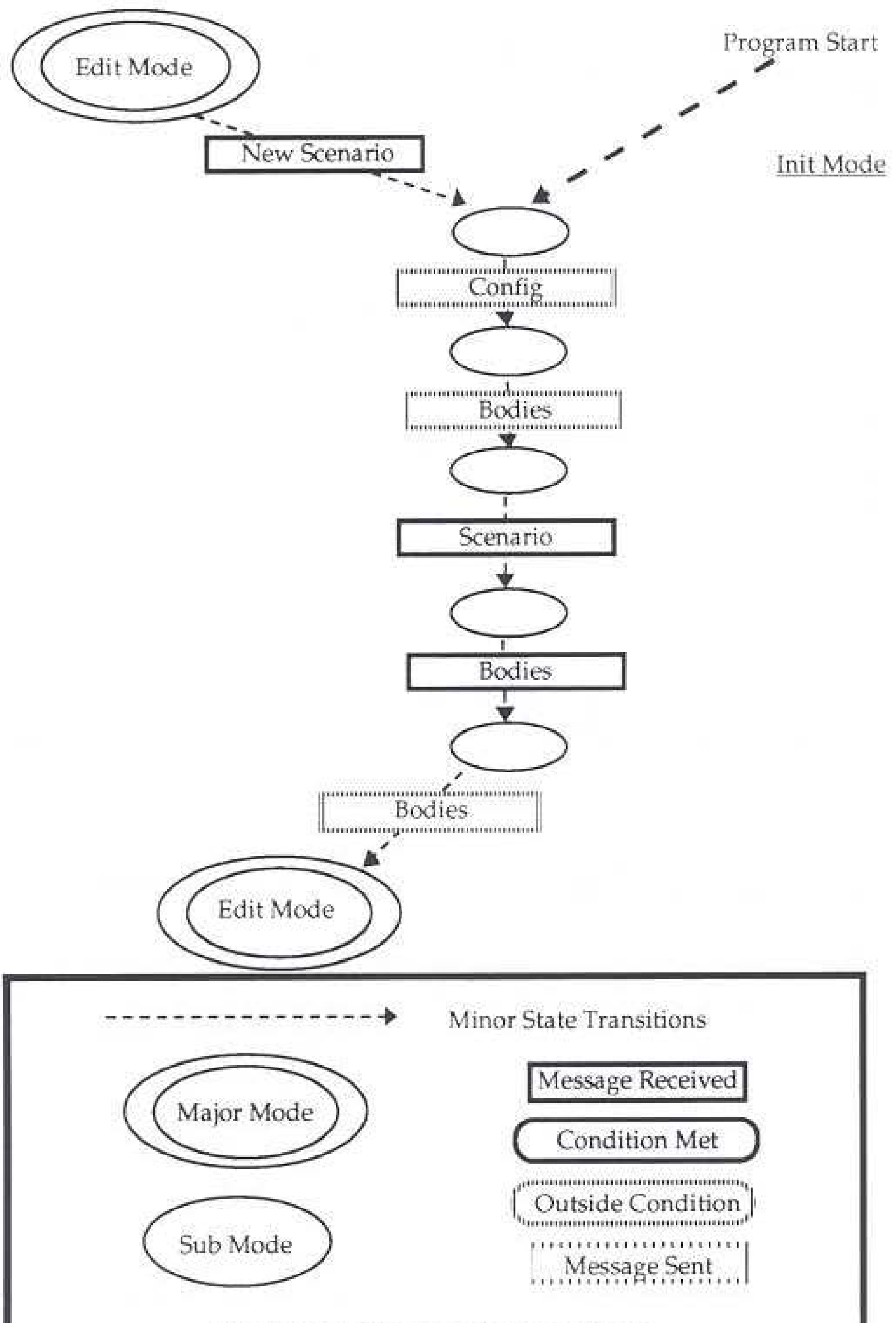


Figure 3.2a Simulation Init Mode

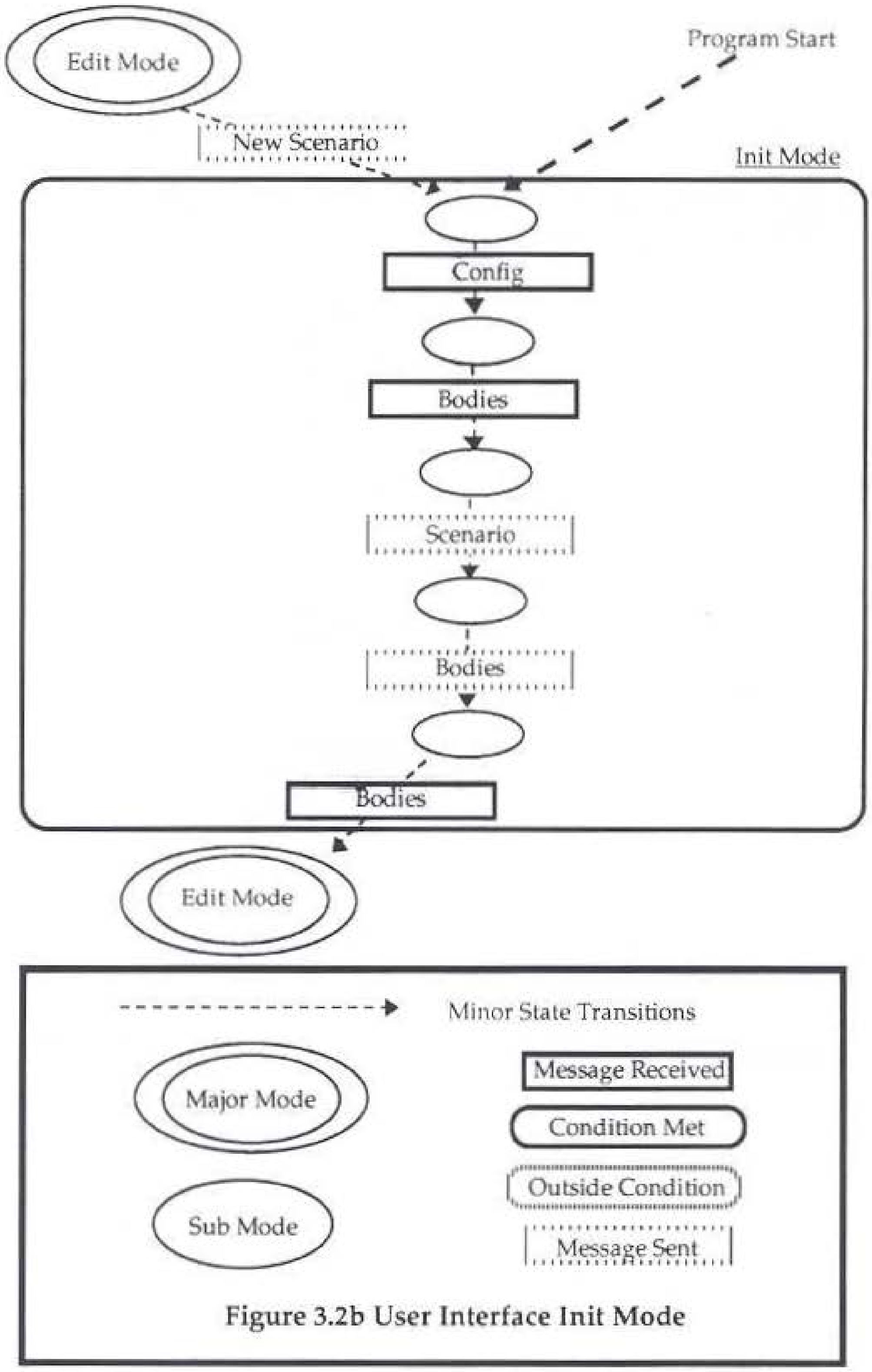


Figure 3.2b User Interface Init Mode

3.3.2. Edit Mode

Once the user interface has entered Edit Mode, either by receiving a BODIES message in Init Mode or a STOP message in Pre-Stop Mode, as shown in Figure 3.3b the user performs his desired edits. These edits can result in a variety of messages being sent to the simulation. The BODIES and SCENARIO messages may be sent in any order and, to leave room for future modification, simulations should be able to receive these commands multiple times. The messages which cause a state transition are START, QUIT, and NEW_SCENARIO.

The BODIES message contains the entire list of bodies. This list indicates the new "state" of the bodies. New bodies are marked as NEW and the simulation is expected to fill in any properties that the user interface is not allowed to change. No indication of deleted bodies is given, nor are changed bodies marked. The simulation is expected to simply read in this body list as its new state. The user interface is trusted not to perform any actions disallowed by the simulation.

The SCENARIO message indicates a change in some global part of the scenario. For information on the fields of the message and their conditional transmission, see the *message format* section below.

START is a parameterless message which changes state to Run Mode on transmit from the user interface and on receipt at the simulation.

When the user interface is ready to terminate the simulation it is expected to send a QUIT message to the simulation. Upon receipt of this message, the simulation cleans up and exits. The user interface may then exit likewise or proceed to initialize another simulation.

The NEW_SCENARIO message is also a parameterless message which changes state to Init Mode on transmit from the user interface and on receipt at the simulation.

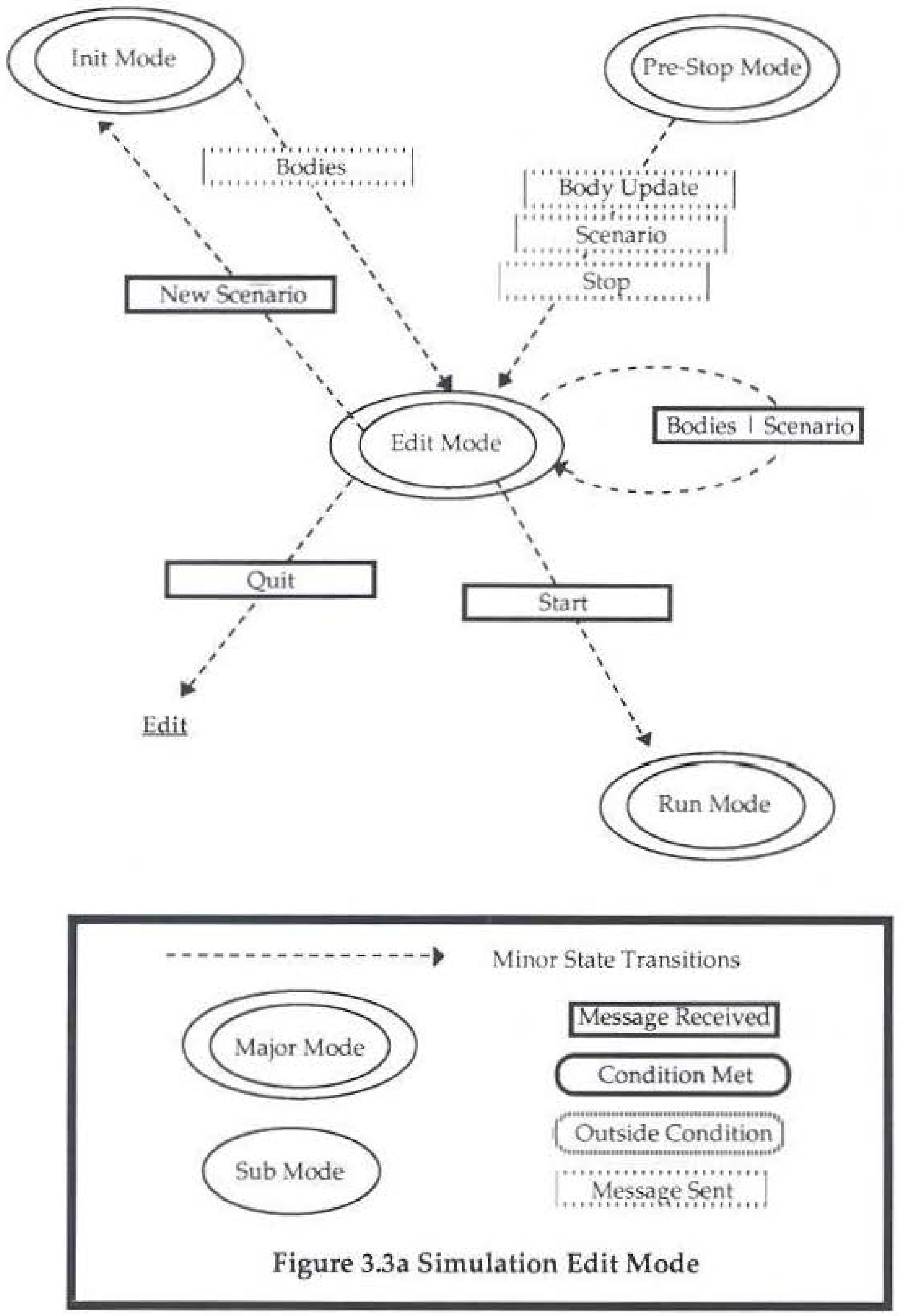
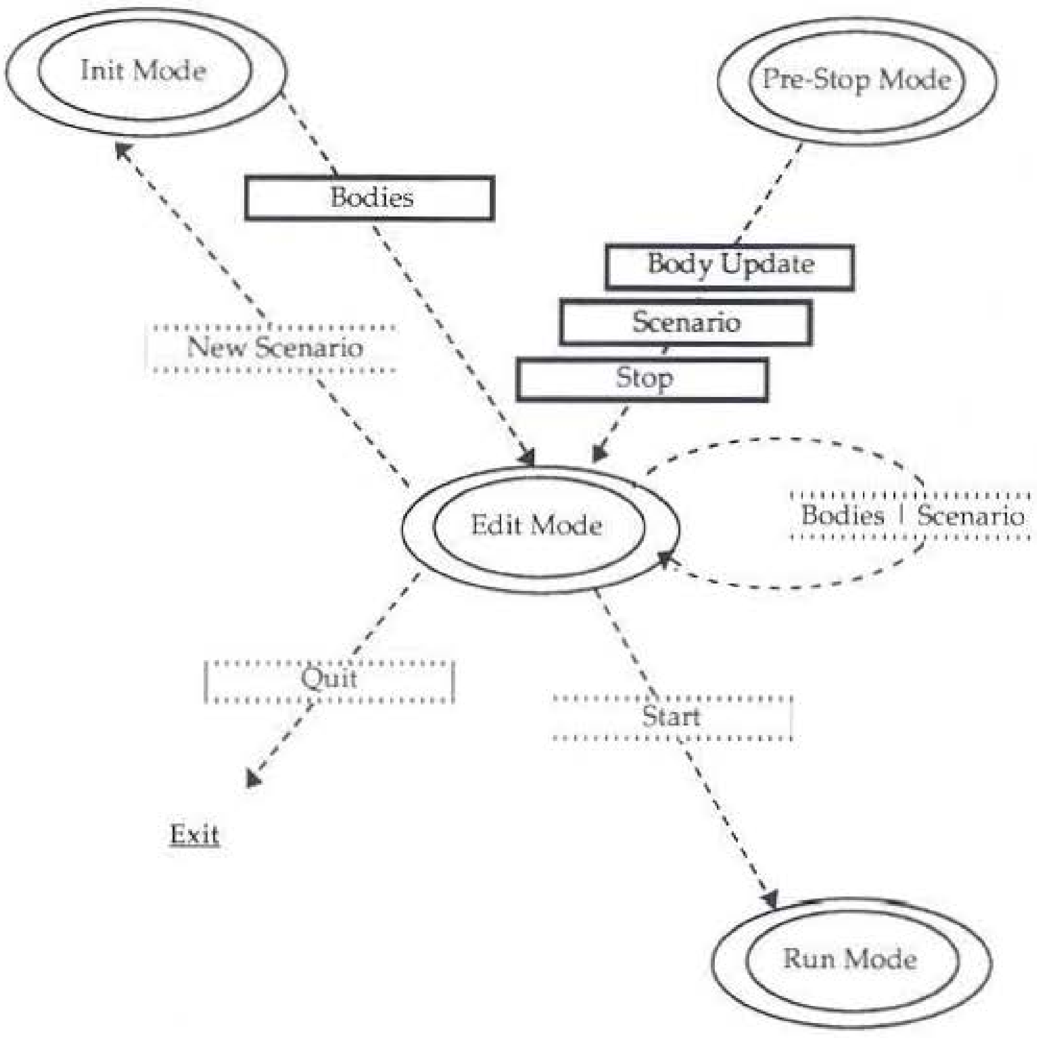


Figure 3.3a Simulation Edit Mode



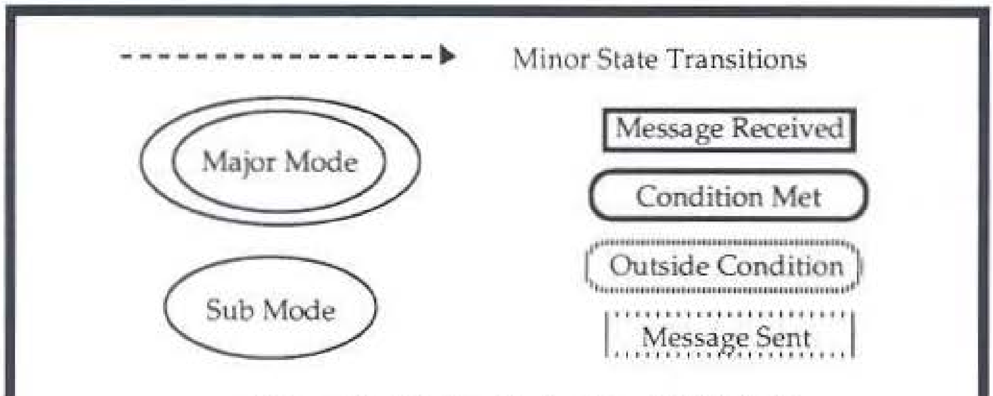


Figure 3.3b User Interface Edit Mode

3.3.3. Run Mode

Run Mode has some of the more complicated actions as seen in Figures 3.4a and b. The POSITIONS message is the only message that can travel from the simulation to the interface while in this mode. The user interface can send THRUST, SAMPLE_RATIO, NO_OP, or STOP messages while in Run Mode.

To update body positions in the user interface, the simulation sends a POSITIONS message after every SAMPLE_RATIO iterations of the current time step in the simulation. (If the SAMPLE_RATIO is less than one, the simulation sends the same frame 1/SAMPLE_RATIO times). To maintain synchronization between the user interface and the simulation, the user interface acknowledges each POSITIONS message. However, to avoid delaying the updates because of network or host load we allow the simulation to continue running with up to N unacknowledged POSITIONS messages in transit. If the cap is ever reached then the next send of a POSITIONS message will block until an acknowledgement is received.

The user interface acknowledges received POSITIONS messages by setting the ACK field in THRUST, SAMPLE_RATIO, or NO_OP messages.

THRUST, SAMPLE_RATIO, and NO_OP messages may only be sent in response to a received POSITIONS message and one and only one of them may be sent per POSITIONS message received. If messages of multiple types are pending they are processed as follows. If a SAMPLE_RATIO message is pending transmission, send it with the ACK field set and disregard any pending THRUST messages. Otherwise, if a THRUST message is pending, send it with the ACK field set. Finally, if neither of the other messages are pending, send a NO_OP (which are never pending) with the ACK field set. In reality, multiple pending messages is a rare case indicating that the user was able to switch between the tools for those messages in the space between the arrival of two POSITIONS messages.

Despite this limitation on the transmission of THRUST, SAMPLE_RATIO, and NO_OP messages, it is still possible for them to get clumped in transit and, consequently, for more than one of these messages to arrive at the simulation during the space of a single frame. As such, the simulation is expected to receive all available messages after each transmission of a POSITIONS message, decrementing its count of unacknowledged messages appropriately. Further, the receipt of multiple THRUST and/or SAMPLE_RATE messages should be treated as if only the last message arrived. The earlier settings should be discarded.

Finally, a STOP message is highest priority and if one is sent, any THRUST or SAMPLE_RATIO messages pending transmission are discarded. A STOP message results in a transition to Pre-Stop Mode in both the simulation and the user interface.

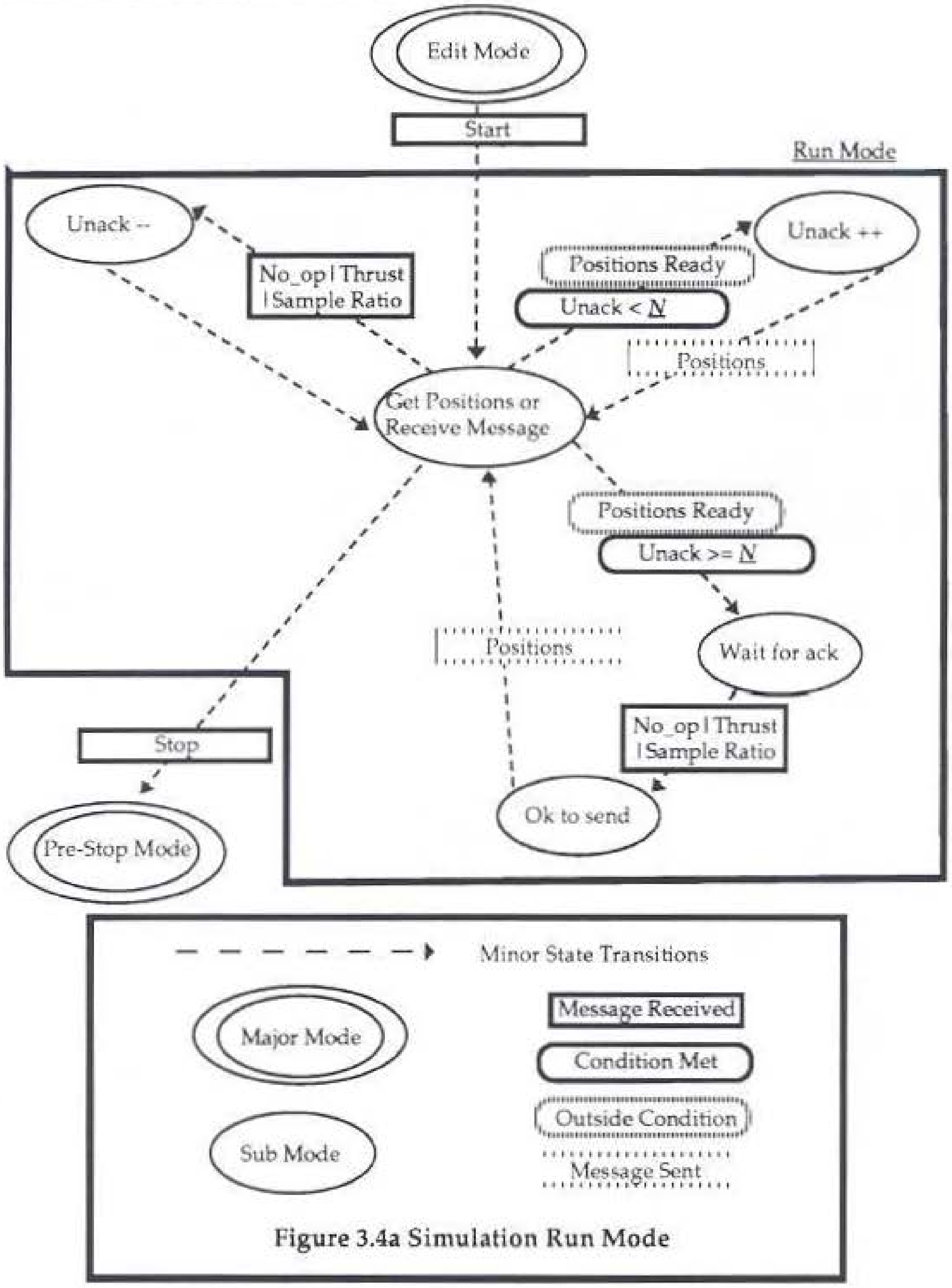


Figure 3.4a Simulation Run Mode

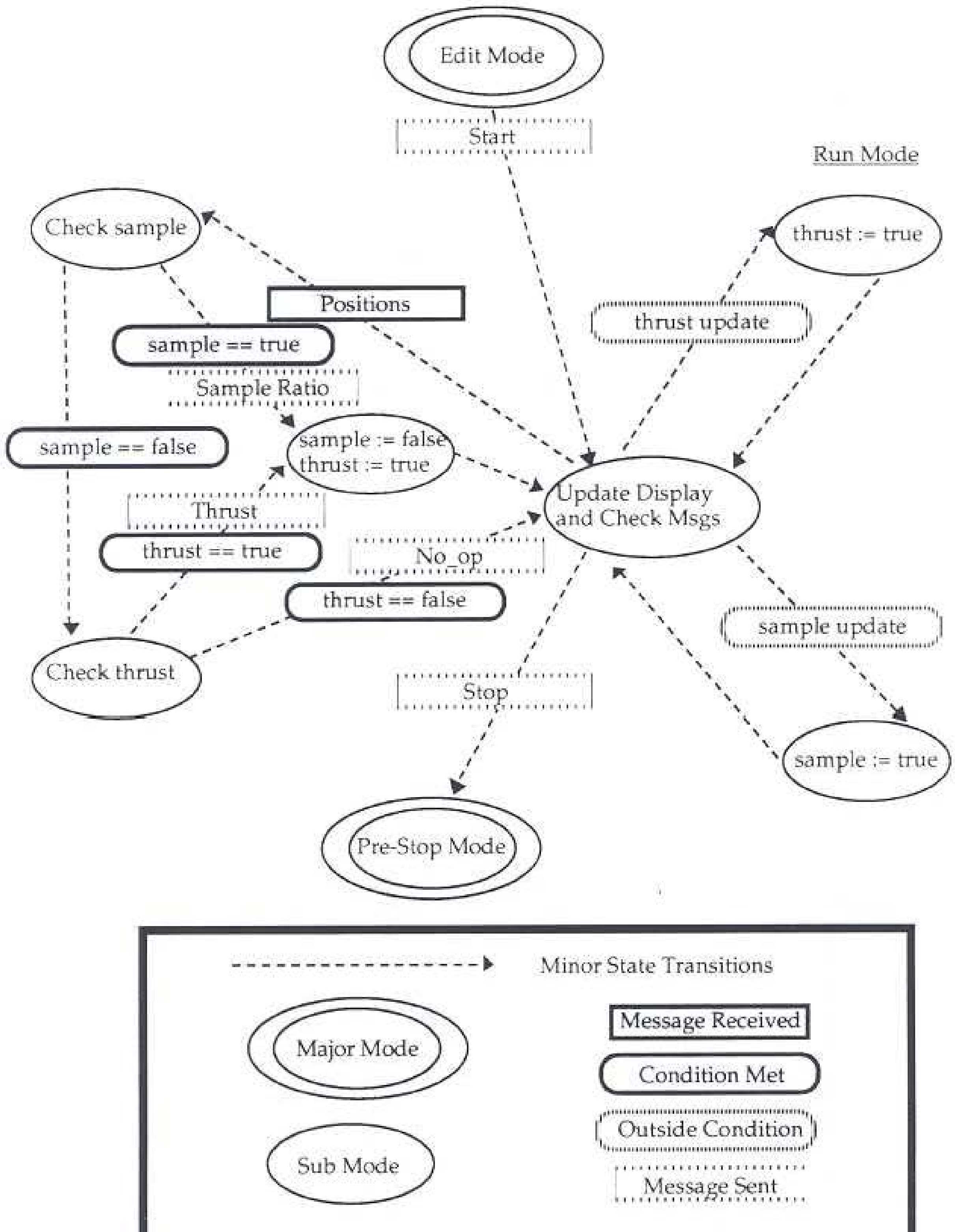


Figure 3.4b User Interface Run Mode

3.3.4. Pre-Stop Mode

As a result of messages in transit as described above, it's possible for a user interface to receive POSITIONS messages after sending the STOP message. As a result this mode supports the messages sequence shown in Figures 3.5a and b. Similarly, it's possible for the simulation to receive acknowledgements after receiving a STOP message. However, THRUST and SAMPLE_RATIO messages are neither sent nor received during this mode. Note that the only messages sent by either side during this phase are the user interface's acknowledgements of the POSITIONS messages that were already in transit when this mode started and the BODIES_UPDATE, SCENARIO, and STOP messages indicating the end of the mode.

This is essentially a catch up stage, allowing both sides of the protocol to become fully synchronized. Towards this end, the user interface continues to receive POSITIONS messages, acknowledging each with a NO_OP with the ACK field set.

The simulation receives NO_OP messages with the ACK field set until its count of unacknowledged messages returns to zero.

Once its count of unacknowledged messages has returned to zero the simulation is expected to signal the end of Pre-Stop Mode by sending a BODY_UPDATE message, indicating the current state of the bodies. The BODY_UPDATE message may carry any body information except the position of the bodies since the user interface has been receiving this information all along. The format of a BODY_UPDATE message and the fields that are expected to be filled in are defined in the *message format* section. Following the BODY_UPDATE message the simulation should send a SCENARIO message and a STOP message to indicated the end of Pre-Stop mode. Actually, the BODY_UPDATE and SCENARIO messages could be sent before receipt of the final ack since TCP guarantees that they will arrive after the POSITIONS message which may still be in transit.

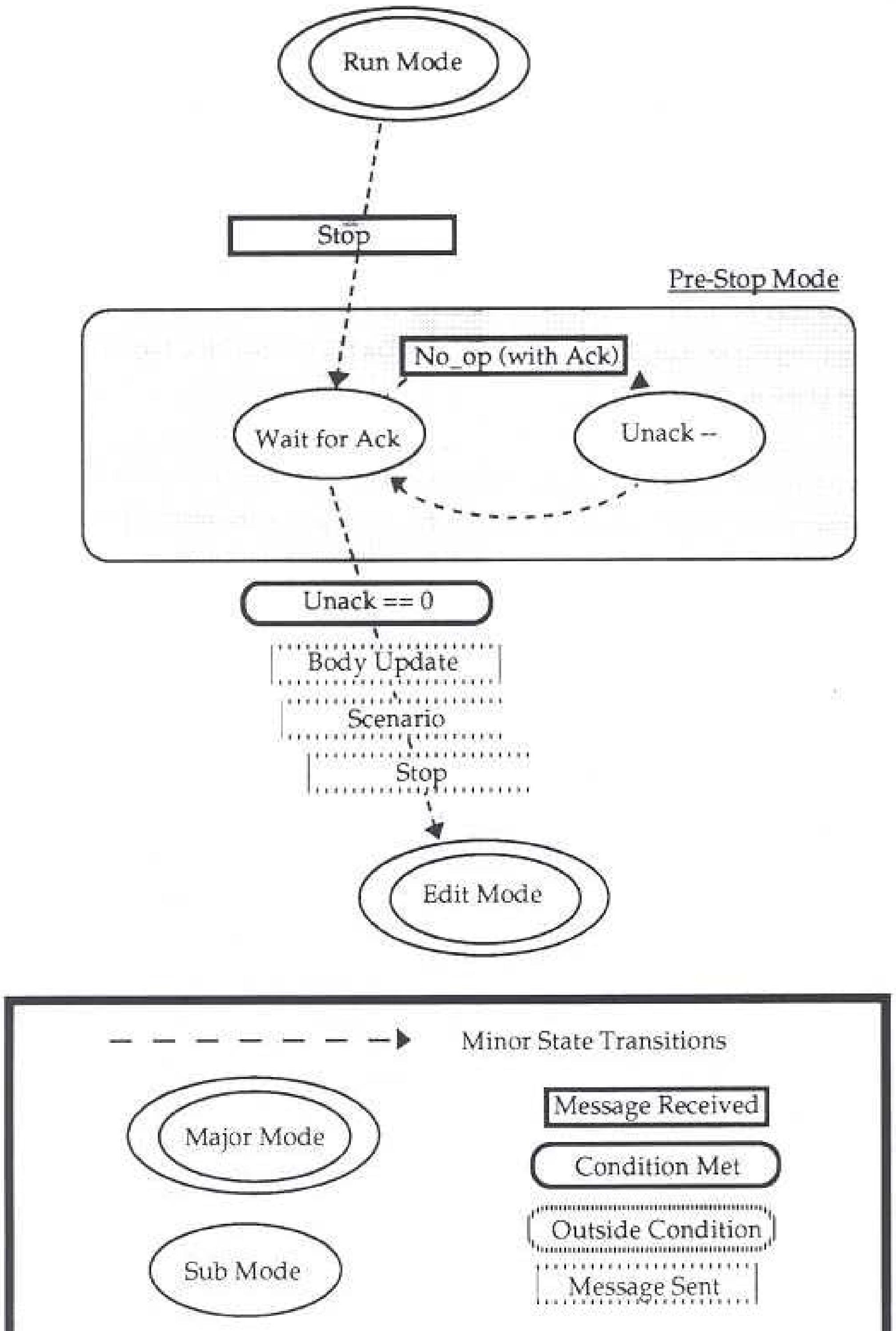


Figure 3.5a Simulation Pre-StopMode

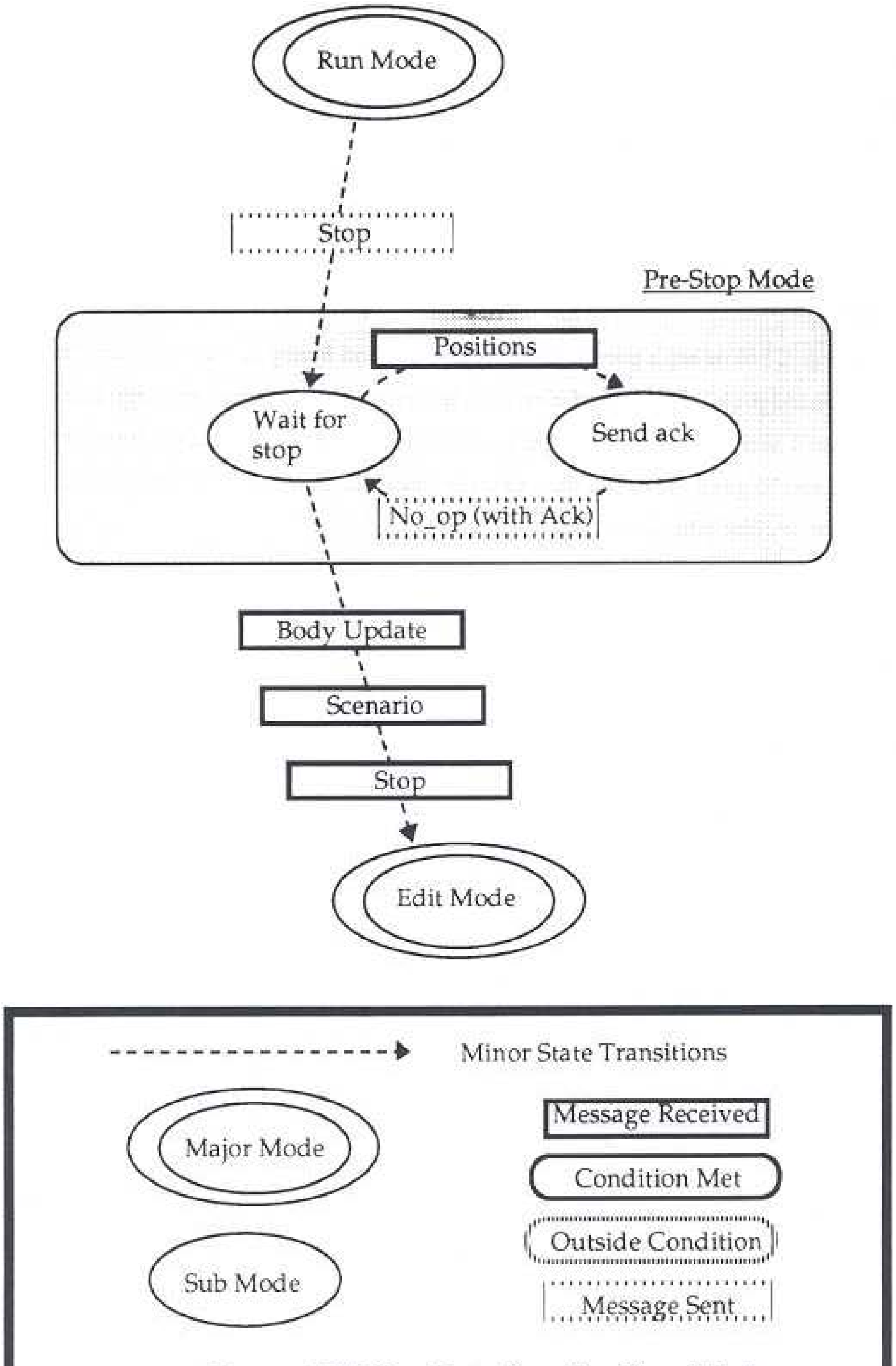


Figure 3.5b User Interface Pre-Stop Mode

3.3.5. Special Messages

In addition to all of the messages described above there is one very important message that may occur at any time: ERROR. An error message may occur in any mode and may be sent to the user interface or the simulator. This message carries with it a code and a string. Interpretation of these codes is ***to be determined***.

3.4. Library routines

The functions described below are a part of libnbp.a and defined in nbp.h. The N-Body Protocol library provides support for the protocol described above, including data conversion, message formatting, simulation startup and sending and receiving the messages in the protocol. In addition to the straight forward support of sending and receiving the messages described above, the library provides support for initializing a remote process (the simulation).

Further a level of data hiding is provided since the protocol library hides the connection, keeping track of the connection internally. In this implementation of the protocol a TCP/IP socket is used but other choices of communication paths could be implemented inside of the library simply requiring programs using the library to relink.

Further, the protocol library hides some of the subtler points of the protocol from the users. This is most obvious in Run Mode. In this mode, the user interface does not have to explicitly acknowledge POSITIONS messages nor does it have to limit the number of THRUST or SAMPLE_RATIO messages to one per POSITIONS message received. When calls are made to nbp_send_thrust or nbp_send_sample_ratio messages are not actually sent, but rather the value passed is stored and a marker is set indicating that the call was made. Later, when a POSITIONS message is received the function checks to see if a marker is set for the sample ratio. If so, a SAMPLE_RATIO message with a piggy backed ACK is sent and the sample ratio and thrust markers are cleared.

If no sample ratio marker is set, the thrust marker is checked. If set, a THRUST message with a piggy backed ACK is sent to the simulation and the THRUST marker is cleared. Finally, if no markers are set, a NO_OP with a piggy backed ACK is sent to the simulation.

On the simulation side of the protocol, the count of unacknowledged POSITIONS messages is kept inside of the protocol. The simulation doesn't have to keep track of them, nor does it have to process acknowledgements. When an acknowledgement arrives it is processed by the protocol library and the count of unacknowledged POSITIONS messages is decremented. Further if the simulation makes a call to nbp_send_positions or nbp_send_converted_positions with an unacknowledged message count of ***N*** then the library blocks the simulation until an acknowledgement is received, then sending the message and returning.

While the simulation doesn't have to handle acknowledgements explicitly, it does have to make a call to nbp_collect_acks in Pre-Stop Mode. This call blocks until all messages have been acknowledged.

The simulation is responsible for making calls to nbp_msg multiple times per frame to make sure that all available THRUST and SAMPLE_RATIO messages are processed.

The library also records the features that are supported by the simulation and formats and unformats messages conditionally based on the library. For example, if the simulation does not enable the send_color feature, color is not sent in any messages, while, if enabled, it is sent from the user interface to the simulation as part of the BODIES message. The effect of features on message formatting is described with each message format description in the Message Format section.

The library also does some checking to avoid obvious errors in the protocol. For example, neither nbp_run, nor nbp_connect may be called repeatedly without an intervening nbp_close. Also, if nbp_msg returns a code indicating a message that requires processing, calling any processing routine accept the expected one results in an error code. This imposes a level of rigidity in the library but should help to avoid programming errors which might otherwise be quite complicated to find.

The library operates on the standard data types described in the Data Types Section of this manual (Section 1.3).

int nbp_run *(user interface only)*
Using the tagged entry in the tag_file (*see Process Configuration File in the User's Manual*), this routine starts the appropriate application on the appropriate machine and establishes a connection with that process. The parameters of this function are described below. Calling this function twice without calling nbp_close is an error, and an error code is returned with no action taken.

`char *tag` - A tag identifying the entry in the specified tag file which contains the program and machine information.

`char *tag_file` - The name of the file to look for the specified tag in.

int nbp_connect *(simulation only)*
After the simulation is initialized this function takes the machine name and port of the remote process (user interface) and initializes a TCP/IP socket connection. This function returns a zero if everything is

ok, otherwise it returns a negative error code defined in `nb_error.h`. The parameters of this function are described below.

`char *machine_name` - The name of the machine to connect to.

`int port` - The port number to connect to on the remote machine.

int nbp_invalid_msg

Passed a file pointer, message code, and mode, nbp_invalid_msg prints an appropriate message to the passed file pointer based on the passed code and mode.

`FILE *fp` - The file to write the message to.

`NbMsgtCodeType nbp_code` - The message that was received.

`NbpModeType nbp_mode` - The mode the application was in when the message was received.

int nbp_error

Passed a file pointer, prefix string, and error_code , nbp_error prints an appropriate message prefixed with the specified prefix string to the passed file pointer based on the passed error code.

`FILE *fp` - The file to write the message to.

`char *prefix` - A string to prefix the message with.

`int error_code` - The code to base the message on.

NbpMsgCodeType nbp_msg

This function checks to see if any messages have arrived and reports the type of the message. If there is an error looking for a message, then error_code is given a number to indicate the type of error and the function returns the value negative one. The parameters of this function are listed below. Note that receipt of an error message does not cause a return code of negative one nor the setting of an error_code, but rather a return code ERROR_MSG. If no messages have arrived (or if an NO_OP acknowledgement has arrived) then NO_MSG is returned.

`int *error_code` - a pointer to the memory location to store the error_code into.

int nbp_send_config *(simulation only)*

After establishing the connection using nbp_connect, the simulation should use this function to send a CONFIG message indicating the functions and laws supported by this process. This function returns a zero if everything is ok, otherwise it returns a negative error code defined in `nb_error.h`. The parameters of this function are described below.

`FeatureInfoType *feature_info` - Contains info on how many bodies the simulation supports in each feature set and the features associated with each. Also includes max bodies, minimum enabled bodies, and the number of bodies in the alpha set.

`ScenarioType *scenario` - the scenario parameters, including laws and their parameters, the parameters' ranges, and the time step and sample rate.

`RangeType *ranges` - The ranges for body and scenario parameters (excluding law parameters which are in the ScenarioType.)

`BodyType *default_body` - A default body containing color and/or radius as needed.

int nbp_rcv_config *(user interface only)*

Called solely by the user interface, this routine receives a message from the simulation identifying features the remote process supports as well as the laws supported. The parameters of this function are described below.

*FeatureInfoType *feature_info* - Is filled with info on how many bodies the simulation supports in each feature set and the features associated with each. Also includes max bodies, minimum enabled bodies, and the number of alpha set bodies.

*ScenarioType *scenario* - filled in with the scenario parameters, including laws and their parameters, the parameters' ranges, and the time step and sample rate.

*RangeType *ranges* - The ranges for body and scenario parameters (excluding law parameters which are in the ScenarioType) as supported by the simulation.

*BodyType *default_body* - A default body to receive color and/or radius if necessary.

int nbp_send_bodies

When sending the full set of body information this function is used. The bodies' positions, velocities, masses, charges, colors, and radii may be conditionally sent to the process on the other end of the socket as described in the BODY message format section below. The parameters of this function are described below. Note that if any of the pointers in the bodies parameter are null, the message is sent without those fields and an appropriate tag (as described in the message format section) is set in the message header indicating such. This is essentially treated as if that set of properties didn't change since they were initialized, allowing simulations that don't care about, for example, charge to avoid allocating space for an array.

`short num_bodies` - The number of bodies to send.

`BodyListType *bodies` - A structure with pointers to the arrays of body parameters.

int nbp_rcv_bodies

After receiving a BODIES MsgCodeType on a call to nbp_msg, nbp_rcv_bodies is called to receive the body information into a set of arrays of body properties pointed at by the BodyListType. As with nbp_send_bodies, any or all of the body properties may be received conditionally depending on the current features and allocation of arrays by the simulation. The parameters of this function are described below.

`short *num_bodies` - The number of bodies received.

`BodyListType *bodies` - A structure with pointers to the arrays of body parameters.

int nbp_send_body_update *(simulation only)*

This function is essentially the same as the nbp_send_bodies but it's a bit more selective in what is sent. Intended for use solely from the simulation during the transition from Pre-Stop to Edit Mode, positions are not sent since they are sent every time a sampling of the simulation is taken during Run Mode. Additionally, color and radius are never a part of this message because the simulation's control over color and radius is limited to specifying the initial colors of bodies in the fixed set and providing a default color and/or radius if the CHANGE_COLOR and/or CHANGE_RADIUS features are disabled. Once the simulation exits Init Mode it is not allowed to manipulate the color or radius parameters.

`short num_bodies` - The number of bodies to send.

`BodyListType *bodies` - A structure with pointers to the arrays of body parameters.

int nbp_rcv_body_update *(user interface only)*

This function is essentially the same as nbp_rcv_bodies but the properties that might be received are a bit more limited. In particular, positions do not arrive with this message since the sole purpose of the preceding Mode (Run Mode) is to continuously update positions via the POSITIONS message. Additionally, color and radius are never a part of this message because the simulation's control over color and radius is limited to specifying the initial colors of bodies in the fixed set and providing a default color and/or radius if the CHANGE_COLOR and/or CHANGE_RADIUS features are disabled. Once the simulation exits Init Mode it is not allowed to manipulate the color or radius parameters.

`short num_bodies` - The number of bodies to send.

`BodyListType *bodies` - A structure with pointers to the arrays of body parameters to be overwritten.

nbp_send_scenario

This function sends the index of the selected law, parameters for that law, the sample ratio, and the time step indicated in the structure pointed to by scenario as appropriate given the feature set. The parameters of this function are described below.

`ScenarioType *scenario` - A pointer to the currently active ScenarioType.

nbp_rcv_scenario

After receiving a SCENARIO MsgCodeType on a call to nbp_msg, nbp_rcv_scenario is called to receive the scenario information. The parameters of this function are described below.

`ScenarioType *scenario` - A pointer to the ScenarioType to update

nbp_quit *(user interface only)*

This function simply sends the parameterless QUIT message to the simulation indicating termination.

nbp_start *(user interface only)*

This function simply sends the parameterless START message to the simulation indicating termination.

nbp_stop *(user interface only)*

This function simply sends the parameterless STOP message to the simulation indicating termination of run mode.

nbp_send_positions *(simulation only)*

This function sends the positions of *num_bodies* in the arrays pointed to by the *bodies* structure. It should only be called by the simulation. This routine may block until an acknowledgement message is received if the internal count of unacknowledged messages is N. In other cases the count of unacknowledged messages is simply incremented when the message is sent. Once it gets past the blocking stage (if any), this routine sends a position message containing the updated body positions, and, if SHOW_VELOCITY is set, the velocities.

`short num_bodies` - The number of bodies to send.

`BodyListType *bodies` - A structure with pointers to the arrays of body parameters.

nbp_send_converted_positions *(simulation only)*

This function behaves exactly like nbp_send_positions except that it accepts a buffer of pre-converted position data to send instead of a list of body positions which are converted internally. The data is appended to a POSITIONS message header and sent.

`short num_bodies` - The number of bodies to send.

`void *buffer` - A pointer to the buffer containing the converted positions. It is assumed to be in the format expected in the POSITIONS message.

`void *v_buffer` - A pointer to the buffer containing the converted velocities. It is assumed to be in the format expected in the POSITIONS message. If NULL, no velocities are sent.

nbp_rcv_positions *(user interface only)*

After receiving a return code of POSITIONS from nbp_msg, the user interface (the simulation never receives position messages) should call nbp_rcv_position to have its BodyList updated with the most current position data..

`short *num_bodies` - The number of bodies received.

`BodyListType *bodies` - A structure with pointers to the arrays of body parameters. The position array is updated.

nbp_send_thrust *(user interface only)*

This routine updates the thrust value to send out with the next acknowledgement. The procedure for deciding when to actually send the thrust message is described in the Run Mode section of the Message Sequence section of the protocol.

`short body_index` - The body to apply thrust to.

`XYZType thrust_vector` - The thrust vector to apply to the body.

nbp_rcv_thrust *(simulation only)*

Upon receiving a return code of THRUST from nbp_msg, the simulation is expected to call this function to receive the latest thrust vector to apply to a simulation body. The parameters for this function are described below.

`short *body_index` - The body to apply thrust to.

`XYZType thrust_vector` - The thrust vector to apply to the body.

nbp_send_sample_ratio *(user interface only)*

Like nbp_send_thrust, this message really only updates a value which may be sent as part of the next acknowledgement. The procedure for choosing when or if to send the sample rate is described under Run Mode in the Message Sequence section of this document. The parameter passed is described below.

`SampleType *sample_ratio` - How often the simulation should sample the position data and send it to the user interface.

nbp_rcv_sample_ratio *(simulation only)*

After being notified of the arrival of a sample_ratio message via the return code from nbp_msg, the simulation should update its sample ratio by calling this function. The parameter is described below.

`SampleType *sample_ratio` - How often the simulation should sample the position data and send it to the user interface.

nbp_collect_acks *(simulation only)*
During Pre-Stop mode the simulation is expected to receive acknowledgements until its count of unacknowledged messages returns to zero. This function serves that purpose by blocking until the libraries internal count of unacknowledged messages returns to zero.

nbp_send_error
In the event of some form of error, a notification may be sent to the other process via the this function. This message may be sent by either process in any mode. The message will be truncated to NBP_ERROR_MSG_SIZE as defined in `nbp.h` as necessary.

`int error_code` - A numeric code indicating the type of error that occurred.
`char *msg` - A message describing the error.

nbp_rcv_error
This function receives the body of an error message, including a code indicating the type of message (as described in the error section of the Message Sequence section of this document) and a string describing the error. This string will not exceed NBP_ERROR_MSG_SIZE as defined in `nbp.h`.

`int *error_code` - A numeric code indicating the type of error that occurred.
`char *msg` - A message describing the error.

nbp_close
To close a connection nothing more than a call to nbp_close is needed. A return code of zero indicates success and codes less than zero indicate an error.

Sample code
Sample simulation and user interface programs are provided in `nbody/src/protocol/test`. The sample simulation is `sample_sim.c` and the sample user interface is `sample_ui.c`

3.5 Message Formats

The formats of the messages used in the protocol are described below. Messages could be categorized in two ways, the mode they occur in or the type of message (i.e. information or control). Both categorizations are indicated in the section titles below.

3.5.1.CONFIG *(Data; Init Mode)* - Figure 3.6(a,b,c,d)

The first byte of the messages is the opcode, with the next byte indicating the number of laws in the simulation. Next are two bytes indicating the maximum number of bodies supported by the simulation followed by two more shorts indicating the minimum number of bodies and the number of alpha set (fixed) bodies respectively. If the number of bodies in the fixed set is zero, then the two bytes for alpha set features and the bytes of pad are not in the message. Next are the features of the beta bodies and world features. Following these items are conditional items for color and radius. If the change_color/change_radius feature is off for either set of bodies then the default color/default radius are sent to the user interface. Next come floating point values representing the ranges of all of the features as well as time step and sample ratio.

Following the ranges comes information about the laws. First a stream of bytes indicating the number of characters in the name of each law and the number of parameters for each law. This stream is padded as necessary to make it a multiple of four bytes long. Next come the defaults for the arguments of each law as well as the ranges of each argument. The primary ordering is by law, with the secondary by argument, with the default first, then the maximum and minimum.

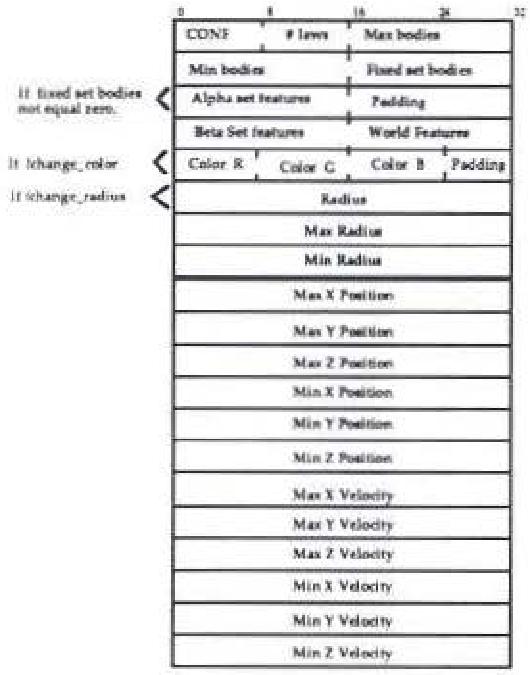


Figure 3.6 a Configuration Message

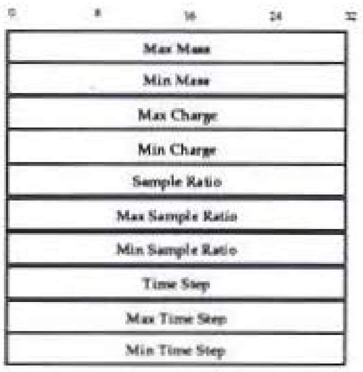


Figure 3.6 b Configuration Message

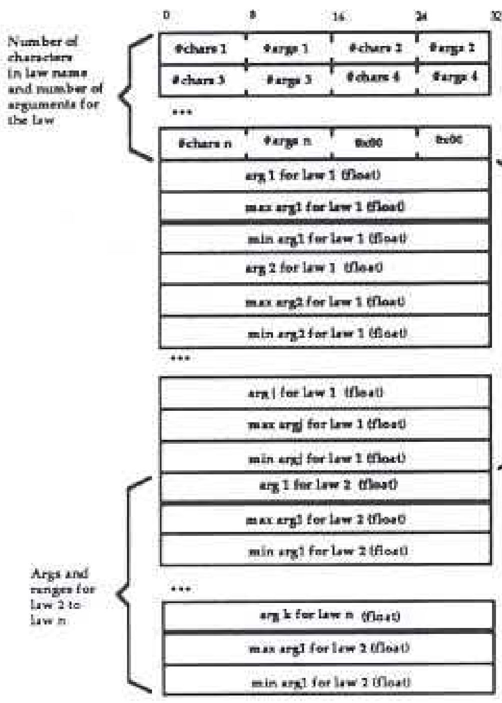

0
8
16
24
32
Number of characters in law name and number of arguments for the law
#chars 1
#args 1
#chars 2
#args 2
#chars 3
#args 3
#chars 4
#args 4
...
#chars n
#args n
0x00
0x00
arg 1 for law 1 (float)
max arg1 for law 1 (float)
min arg1 for law 1 (float)
arg 2 for law 1 (float)
max arg2 for law 1 (float)
min arg2 for law 1 (float)
...
arg j for law 1 (float)
max argj for law 1 (float)
min argj for law 1 (float)
Args and ranges for law 1
arg 1 for law 2 (float)
max arg1 for law 2 (float)
min arg1 for law 2 (float)
...
arg k for law n (float)
max arg1 for law 2 (float)
min arg1 for law 2 (float)
Args and ranges for law 2 to law n

Figure 3.6 c Configuration Message

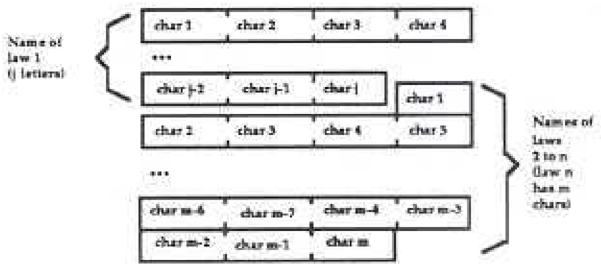


Figure 3.6 d Configuration Message

3.5.2. SCENARIO *(Data; Init, Pre-Stop, Edit)* - Figure 3.7

The scenario includes the opcode for the message, the number of laws available, the currently selected law, and the number of arguments for the currently selected law. Each of these fields is a single unsigned character. Following these items are two four byte floats for time step and sample rate. Next are four byte floats for each of the parameters of the currently selected law.

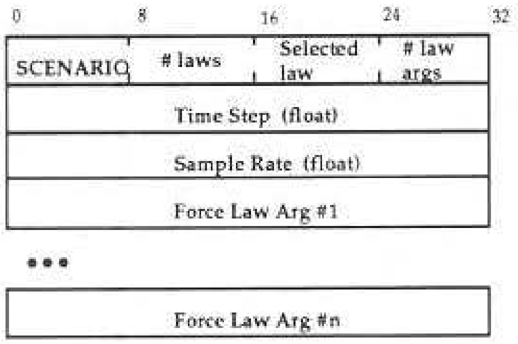


Figure 3.7 Scenario Message

3.5.3. BODIES *(Data; Init Mode, Edit Mode)* - Figure 3.8(a,b)

The Bodies message contains an update on the current status of all of the bodies. The first word consists of one byte for the opcode, one byte for the properties, and two bytes to indicate the number of bodies. The properties field is a bit mask indicating which of the body properties are included in this message. The properties include position, consisting of three four byte floats; velocity, also three four byte floats; masses, one float per body; charge, one float per body; state, one byte per body with padding bytes added to make the total number of bytes a multiple of four; radius, consisting of one four byte float per body; and color with three bytes for red, green, and blue, padded with an additional byte for alignment. The definitions that define the properties field are defined in `nbp.h`.

The determination of whether or not to send a particular property is based both on whether or not the sender is the simulation or the user interface as well as the current settings of the features. However, the format of the message can always be determined from the properties field and as such it should be used in parsing the message in preference to any assumptions about expected information.

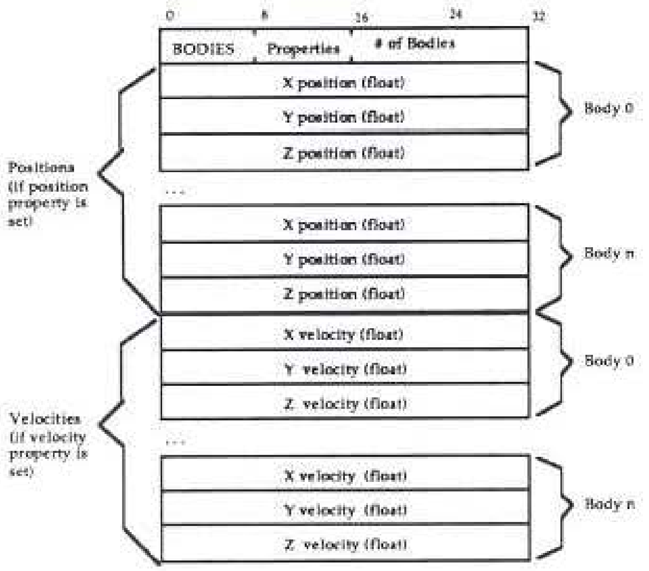


Figure 3.8a Bodies Message

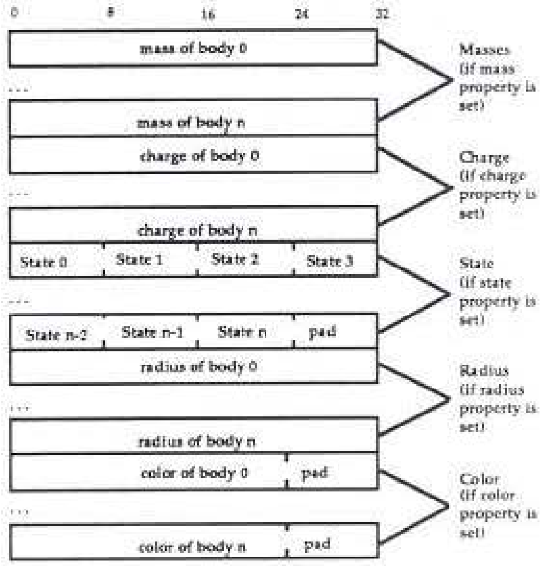


Figure 3.8 b Bodies Message

3.5.4. POSITIONS *(Data; Run Mode)* - Figure 3.9

The positions message is essentially a special case of the bodies message. Always sent from the simulation to the user interface, this message always has the position property and possibly the velocity property as well. The inclusion of the of the velocity information is determined by the setting of the show_velocity feature. If it is on for either set of bodies, velocities are sent.

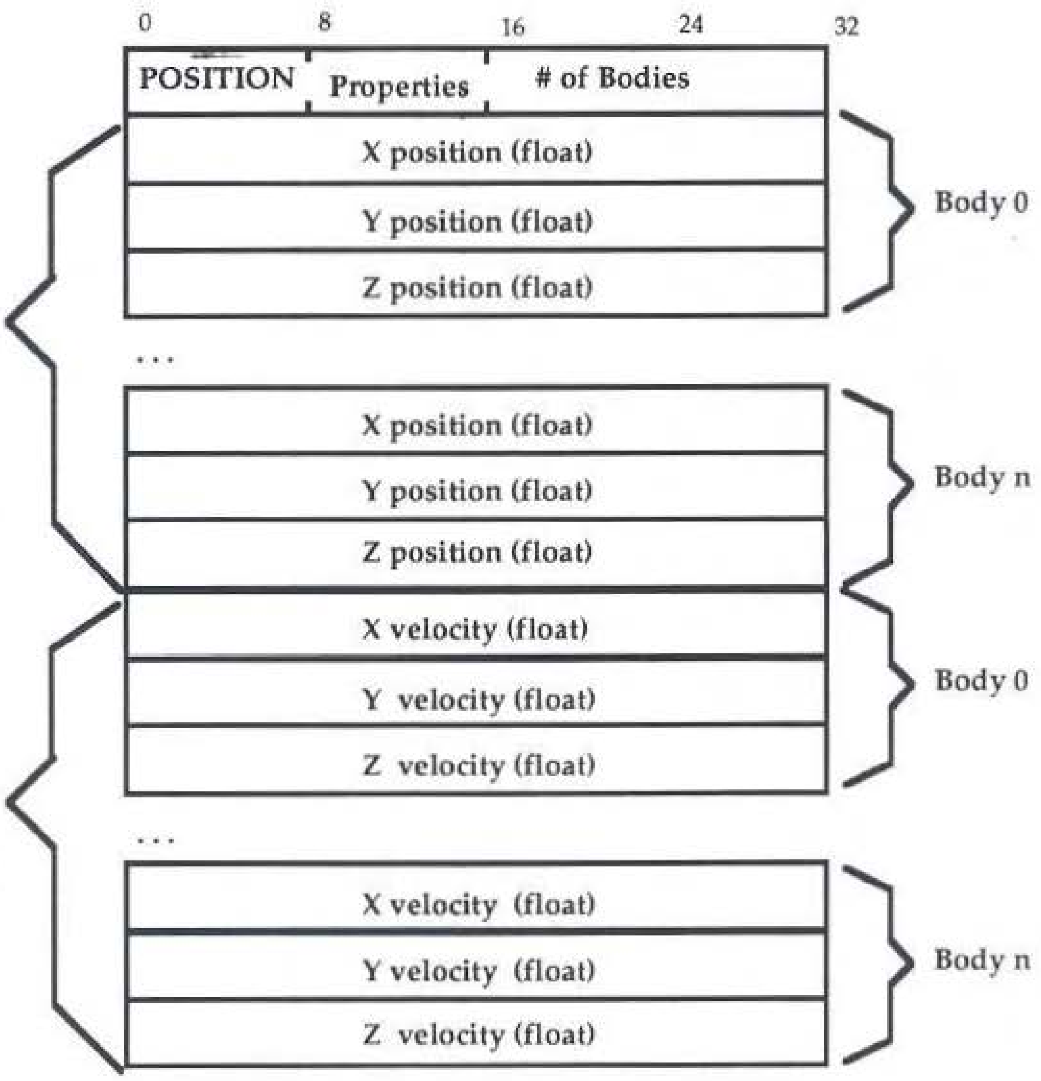


Figure 3.9 Position Message

3.5.5. BODY UPDATE *(Data; Pre-Stop Mode)* - Figure 3.10a,b

Like the positions message this message is always sent by the simulation to the user interface. It's very similar to the bodies message but is guaranteed to never include status, color, or radius information. All of the other fields are sent if the simulation keeps that information for the bodies. As always, the properties should be used to determine exactly what was sent.

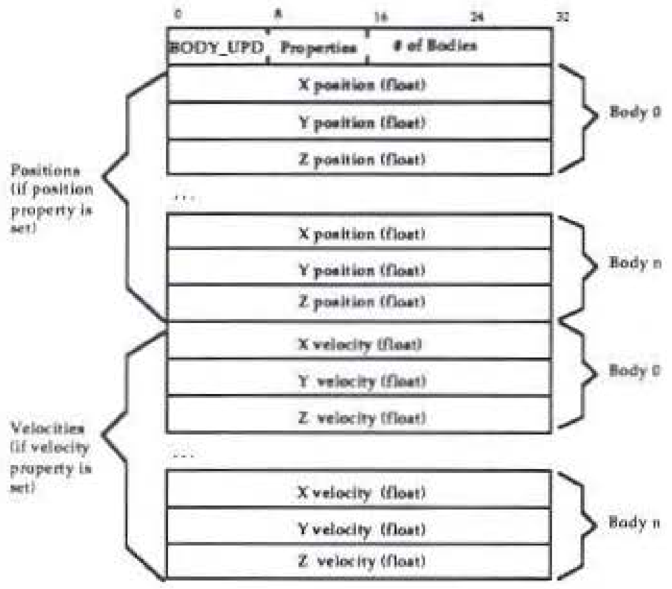


Figure 3.10 a Body Update Message

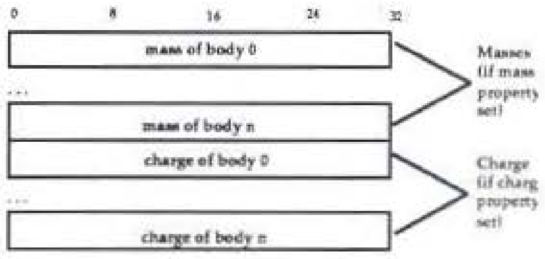


Figure 3.10 b Body Update Message

3.5.6. NEW_SCENARIO *(Control; Edit Mode)* - Figure 3.11

When the user interface loads a new scenario it sends a one byte message to the simulation indicating that both processes should return to init mode and exchange configuration information. This is essentially the same as restarting the simulation with a new scenario, only it is faster. No state from the previous scenario should be kept.

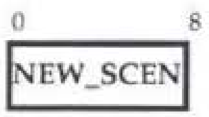


Figure 3.11 New Scenario Message

3.5.7. QUIT *(Control; Edit Mode)* - Figure 3.12

This one byte message, always sent from the user interface to the simulation, is simply an opcode telling the simulation to exit.

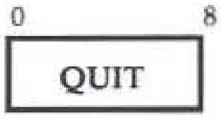


Figure 3.12 Quit Message

3.5.8. START *(Control; Edit Mode)* - Figure 3.13

This one byte message, always sent from the user interface to the simulation, tells the simulation to switch from Edit Mode to Run Mode.

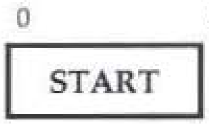


Figure 3.13 Start Message

3.5.9. THRUST *(Data; Run Mode)* - Figure 3.14

This message contains the same header as all of the other Run Mode messages except Stop, consisting of a one byte opcode, an optional (though currently always used) Acknowledgement field, two padding bytes, and a four byte unsigned integer containing the frame number last received by the user interface. In addition to the standard header the thrust message contains a two byte unsigned integer for the body to apply thrust to, two pad bytes, and three four byte floats containing the X, Y, and Z components of the thrust vector.

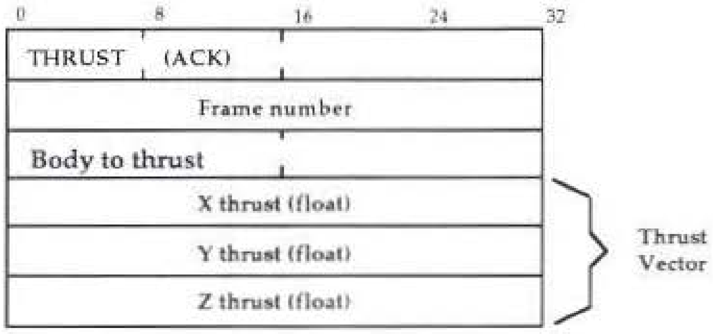


Figure 3.14 Thrust Message

3.5.10. SAMPLE_RATIO *(Data; Run Mode)* - Figure 3.15

In addition to the standard run mode message header described in the Thrust section, this message has a four byte float for the sample ratio.

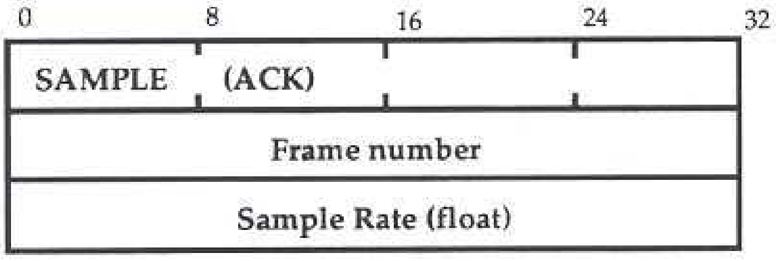


Figure 3.15 Sample rate Message

3.5.11. NO_OP *(Control; Run Mode, Pre-Stop Mode)* - Figure 3.16

The no_op message is simply the standard run mode header described in the Thrust section above with a No_op opcode and an acknowledgement turned on.

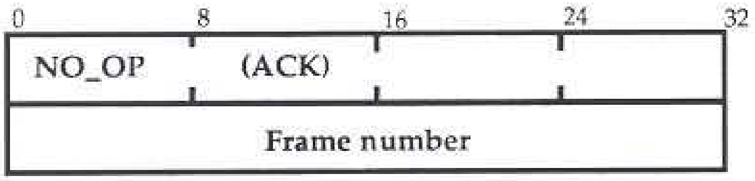


Figure 3.16 Acknowledgement Message

3.5.12. STOP *(Control; Run Mode, Pre-Stop Mode)* - Figure 3.17

Sent by the user interface to exit Run Mode and the simulation to terminate Pre-Stop mode, this one byte message is simply the stop opcode.

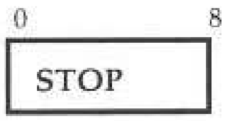


Figure 3.17 Stop Message

3.5.13. ERROR *(Data/Control; Any Modes)* - Figure 3.18

The error message contains in its header a byte for the opcode, an unsigned one byte integer error code, a pad byte and an unsigned byte indicating how many characters there are in the message body. The message body simply contains the error message as specified in the fourth byte of the header.

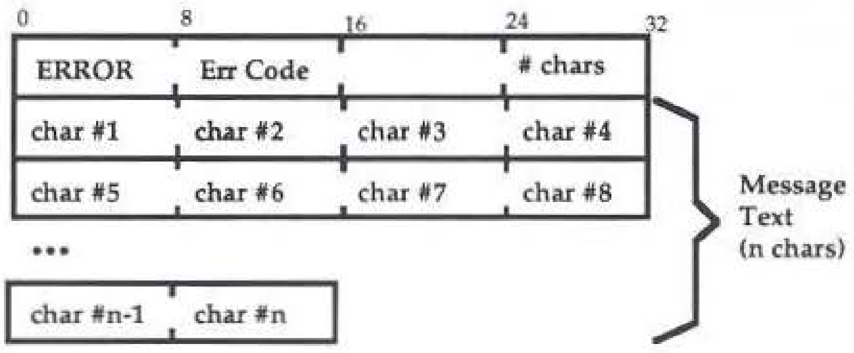


Figure 3.18 Error Message

3.6 Data Conversion

With the exception of nbp_send_converted_positions all data conversion takes place within the library. The protocol transports all data in Big Endian order as used on Suns, also network byte-order. The standard for floating point is IEEE, as available on the Sun. For most of our data conversion we use routines written by Russell Taylor for his own work. These are provided in `libconvert.a` on `/glycine/grip3/arm/dock/communication/lib/$(HW_OS).dir` where `$(HW_OS)` is set to the output of the `hw_os` program on the architecture in question. However, to simplify our tasks, and because of a minor inconsistency in the library, we have a set of macros which hide the calls to `libconvert.a`.

These macros are all contained in `nbp_convert.h` and take a pointer to a character buffer, a pointer to the data type to be loaded, and an integer to increment by the number of bytes added to or taken from the character buffer. Two example macro calls would be `nbp_put_char( out_buf, &ch, buffer_size )` and `nbp_get_char( in_buf, &ch, buffer_size)`. In these examples *ch* would either be put into or gotten out of the buffer. In both cases the buffer pointer would be incremented by the size of a character as would `buffer_size`.

Note that these routines include macros to convert floats to the canonical format as well.

3.7. Support libraries

`Libnbp.a` depends on `libsdi.a` and `libconvert.a`, libraries written by Russell Taylor for starting process on remote machines and establishing connections to them and for data conversion respectively. His work made the implementation of this library much simpler than it might have been otherwise. Both sets of libraries work for sparc, vax, and mips architectures, making running simulations on the department's different machines very easy.

3.8. Future Directions

There are a wide variety of future directions for the protocol from the addition of features to performance analysis and considerations of portability. Some of the ideas are discussed briefly below.

Adaptive window sizes

One possible feature to add to the simulation would be adaptive windows on the number of unacknowledged messages in the network which would grow (or could be initially set) to large sizes during non-interactive runs and automatically reduce when the user interface is sending message indicating a high degree of interactivity such as thrusting. This should be easy to add but determining the algorithm to use for adapting could be difficult.

Performance Analysis

How many bodies can we send smoothly? Where are the bottle necks? What is the lag on interaction with the simulation? How far ahead can the simulation be before the lag in the system is intolerable? All of these questions need to be analyzed.

Design Rationale

We tried to design this protocol with a set of goals in mind. First and foremost we wanted to be able to support a wide variety of simulations easily without requiring the simulation writer to support arbitrary features which were of no real concern to him or her.

Sample simulations

We used several types of simulations as basis for evaluation as we were designing. They are described below. (Note that we never planned to implement all of these simulations, we simply wanted a design that could support them all.)

N-body simulation - While this simulation demands almost all of the available features of the user interface, it is many ways the simplest. In this simulation all bodies are treated equally; the same force laws can be applied to all bodies and the same properties can be changed. Bodies can be added, deleted, repositioned and have any of their parameters modified. The simulation does not care about the color or radius of bodies, but does allow the user interface to change both.

Ephemeris - This simulation lies at the opposite end of the spectrum from the N-body simulation. All bodies continue to be treated equally. Body positions are computed by parametric equations and, since there are a set number of equations, the number of bodies in the simulation is fixed. The simulation does not allow addition or deletion of bodies, nor changing position or velocity. It does allow changing body color and radius, but doesn't care what they are.

Ephemeris and N-Body combined - This simulation presents the most complex problems, combining features of both of the above. In this case, a set number of the bodies represent the planets and, like the planets in the ephemeris simulation above, they are are immune to changes to position or velocity as well as addition and deletion. However, there may be other bodies in the simulation which may be modified, added and deleted. These bodies would be affected by standard force laws, particularly the strong gravities of the planets.

Color sort - This simulation simply sorts bodies based on color, adjusting their positions to indicate their current location in the sort. The unique feature of this simulation is its need to know the color of the body.

Colliding N-bodies - This simulation would be a standard n-body simulation with the ability to detect body collisions and have bodies bounce off of one another. The significant addition here is the simulation's need to know the radii of the bodies.

Problems that we have to address - feature interaction.

Here we discuss some of the problems these simulations present.

Unchangeable properties - Simulations are going to want to control which properties they want the user to have control over and which properties they want to restrict. The N-body simulation allows addition and deletion of properties as well as changing all of their properties. An ephemeris simulation only allows color and radius to change, with the number of bodies remaining constant.

Different features for different bodies - The combined N-body and ephemeris simulation allows the user to add, delete, and change the positions of the additional bodies, but not for the "planets".

4. Overview of Simulation on the Maspar

Quan Zhou

The N-body simulation on Maspar MP-1 is a simulation that combines two simulations in one: an nbody interaction simulation and a fixed orbital bodies simulation. The nbody half may use either gravity or charge interaction laws. The orbital half has a number of bodies with fixed orbits determined by parametric equations. Neither addition, deletion nor changing position or velocity of these bodies are allowed. Bodies from the nbody simulation may be modified and be affected by other bodies including the orbital bodies by force interaction laws. We can observe many interesting body movements using this simulation.

4.1 Features and Functions Supported

The simulation simulates two sets of bodies. They have different features enabled.

- alpha bodies

  The alpha set bodies have fixed circular or elliptical orbits. Each alpha body's movement is fixed by its parametric position equation. The parameters associated with each body's orbital equation will be read from a special configuration file when the simulation is started. The user can specify his own configuration file name. The first line of the configuration file specifies the number of total alpha bodies (could be zero). Each later line specifies an alpha body's mass, the central point of its orbit, the three radius parameters of the orbit's equation and the initial rotational degree. Comment lines are started with "#".

  The simulation doesn't allow the user interface to change the fixed orbit of alpha bodies, but the following features are enabled on user interface side:

  - disable an alpha body
  - change the radius of an alpha body
  - change the color of an alpha body

- beta bodies

  The beta set bodies simulate general many body interactions. Their motions are decided by the nbody interaction law. Each body's movement is affected by its interaction with other bodies including alpha bodies. In other words, the beta bodies react to the forces of all the other bodies (alpha and beta combined), while the alpha bodies are only affected by their own fixed orbital equations.

The simulation allows the following features to be enabled on the user interface:

- add a new beta body
- delete a beta body
- change the mass of a beta body
- change the charge of a beta body
- change the position of a beta body
- change velocity of a beta body
- apply thrust to a beta body
- change the radius of a beta body
- change the color of a beta body
- show velocity

For the scenario features, the simulation enables following:

- change the simulation time step
- change the simulation sample ratio

4.2 Overview of Implementation

Since we are using a massively data parallel processing system to do the simulation, the simulator consists of two parts:

- The front-end control process

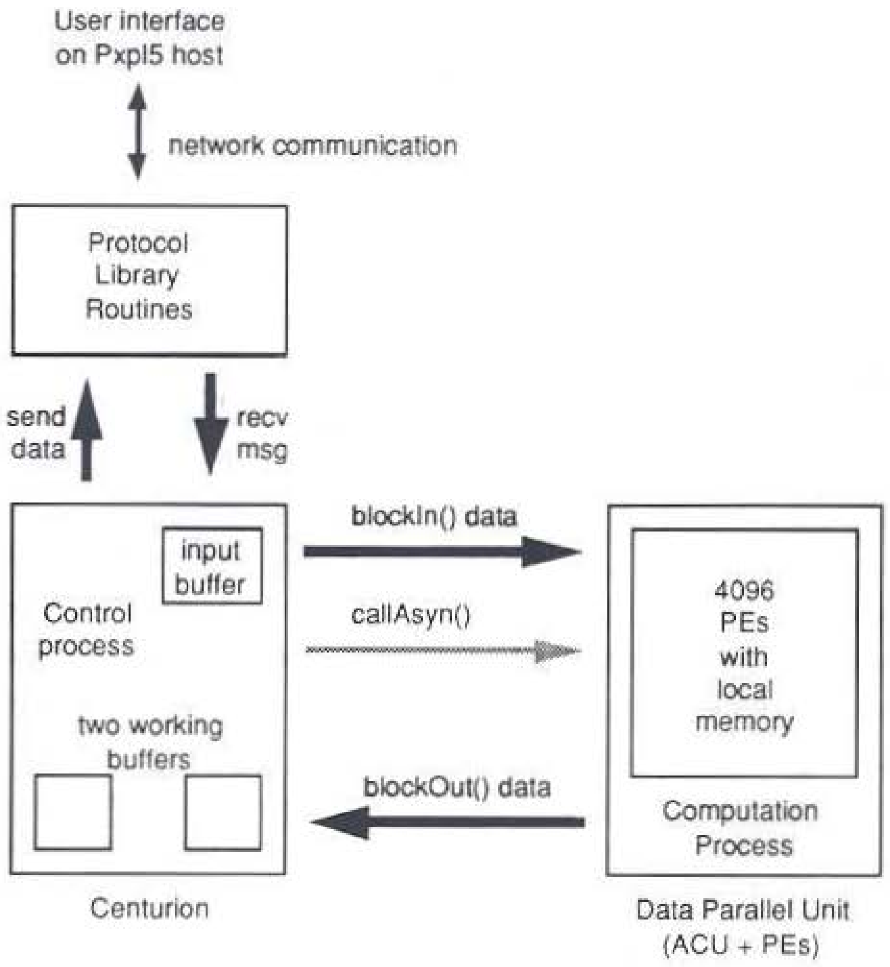
  Running on the front-end machine named Centurion, this process controls the whole simulation procedure. Through protocol messages described in section 3, it receives messages such as body and scenario information from the user interface, directs the back-end DPU to fulfill the required simulation computations in parallel, and sends the simulation results back to the user interface for graphical display at interactive rates. The control process is also responsible for managing the front-end working buffers so as to achieve asynchronous execution between back-end simulation and user interface. The front-end control program is written in the standard C language.

- A back-end computation process

  Running on the back-end DPU (Data Parallel Unit) which consists of an ACU (Array Control Unit) and 4096 PEs (Processor Elements), this process does the actual simulation computations in parallel(SIMD). These computations involve the calculation of body interactions based on the force laws and the update of bodies' positions and velocities for both alpha and beta bodies. The simulation supports two body interaction models: the universal gravity model and the electrostatic charge model. The numerical approximation methods for solving differential equations are employed to simulate the body interactions and movement. The back-end simulation computation program is written in the low-level data parallel language MPL.

These two processes work asynchronously under a producer-consumer model. That is, the simulation computation process on the back-end produces new position and velocity data which are then sent to the control process. The control process then invokes the N-body system's protocol library routine to send the data through the network to the user interface.

The following picture provides a view of communications between simulation processes:



The front-end control process calls the back-end computation process to do the simulation repetitively in an asynchronous fashion. The front-end maintains two working buffers to achieve the asynchronous execution of the two processes, as in producer-consumer model. The two working buffers are used to store the updated positions and velocities data produced by back-end process. New positions and velocities data are produced and blocked out to front-end working buffer after each sample ratio of iterations at the specified time-step. In this way, when the back-end simulation process blocks out data to one working buffer, the control process can call N-body protocol library routines to send out updated data from another working buffer (which has the last output data) to the network.

4.3 Control Process

After reading the initial alpha body configuration and setting the features that this simulation supports, the control process starts in Init mode in which it will achieve synchronization with the user interface. Then it gets into Edit mode and may transfer to Run mode and Pre-stop mode and back to Edit mode again depending on the user interface messages received. The following are the four modes that simulation may get into:

- Init mode

  In Init mode, the simulation sends the initial scenario and alpha, beta body configuration to the user interface. It also tells the user interface which features (separately for alpha and beta bodies) that this simulation enables.

- Running mode

  When the front-end control process receives a "start" message from the user interface in editing mode, it gets into running mode. In the running mode, the control process starts the back-end simulation process to do computations asynchronously. The control process then periodically checks the message from both back-end and network. If there are no messages from user interface, the control process will repetitively request that the back-end process continue the simulation and call the network routine to send out new "positions and velocities" messages (including both alpha and beta bodies) to user interface. During this period, the control process is responsible for managing the front-end working buffers and synchronizing the back-end and front-end when all the working buffers are filled by back-end but are still waiting to be sent to the user interface. If the control process receives messages such as application of "thrust" or "change of sample-ratio" from user interface, it will inform the back-end computation process to adjust appropriately in its next-step computation. When the control process receives a "stop" message from the user interface, it stops calling the simulation process, sends out all the remaining position and velocity data, exits the running mode and gets into pre-stop mode.

- Pre-stop mode

  The Pre-stop mode is a catch-up stage as specified in the protocol. During this period, the simulation will keep sending out remaining "positions and velocities" messages and receiving acknowledgements until both the simulation and the user interface agree to get into editing mode.

- Editing mode

  When the front-end process receives a "stop" message from user interface, it gets into editing mode. In the editing mode, the front-end control process can receive scenario editing messages with updated information and get a new package of body data after the user adding or deleting bodies or changing the body properties. When the front-end process receives "start" message again, it requests that the back-end re-distributes the body data on different processors properly. Especially, since the alpha bodies can affect beta bodies by force law, the back-end needs to adjust the alpha bodies distribution according to changed number of beta bodies so as to achieve efficient computation. Then it exits editing and gets into running mode again.

4.4 Computation Process

The back-end computation process always waits until the front-end control process requests it to do computations asynchronously. Generally, when it is called, the computation process works in the following steps:

1. Calculate the new positions and velocities using the specified body interaction model and the iteration time-step. The iterations will be repeated based on the sample-ratio specified by the control process.

   - For the alpha bodies, we need to compute the next position based on each body's orbital equation. The updated alpha bodies' positions will be sent to the proper place so that they can be used to affect the computation of beta bodies next positions and velocities.

   - For the beta bodies, we will use the numerical approximation method, as explained later, to calculate their updated positions and velocities. Since the simulation simulates the bodies interactions by force laws, the same force law is applied on all beta bodies and all beta bodies are treated equally. We can utilize the SIMD computation to do this work efficiently This is explained later in the "parallel computation" section.

2. In parallel convert the VAX floating point representation to IEEE standard floating point representation that is used by Pxpl5.

3. Block out the new body positions and velocities (by combining alpha bodies and beta bodies) to the specified front-end working buffer.

The following discussion illustrates the important ideas involved in each step above.

4.4.1 Body Interaction Model

The simulation process supports two body interaction models. That is, it provides different force calculation functions for different models. The calculation of the body accelerations are based on the chosen interaction model. Here is a list of interaction models available in this simulation:

• Universal Gravity Law

F = g (m1*m2) / r^2

g - gravity constant

m1,m2 - masses of two interacted bodies

r - distance of two interacted bodies

The direction of force F is the same as that of distance vector.

• Electrostatic Charge Law

F = 1 / (2*PI*c) * (c1*c2) / r^2

c - charge law constant

c1,c2 - charges of two interacted bodies

r - distance of two interacted bodies

The direction of force F is the same as that of distance vector.

4.4.2 Body Placement in PEs

In order to compute body interactions in parallel, we need to map the bodies' data (including masses, positions and velocities) to processor elements. In this simulation, the mapping of body data uses the hierarchical domain decomposition model. That is, neighboring bodies are placed in the same processor. In our setting, body data are distributed along the first row (64 processors) of the Maspar's square array of processors using hierarchical model.

Suppose we have 67 total bodies in which the first 27 are alpha bodies and next 40 are beta bodies, then the distribution of bodies on the first row is as follows:

```
proc: #0 #1 #2 ................................ #32 #33 #34 ....................................... #63

body: #0 #2 #4 ................................ #64 #66  X  ....................................... X
body: #1 #3 #5 ................................ #65  X   X  ....................................... X
```

As we mentioned earlier, the alpha bodies can affect the movement of beta bodies, but not the reverse. In order to adapt to the changed number of beta bodies during the editing mode, the back-end needs to re-distribute both alpha and beta bodies placement in PEs. This is necessary for efficient communication while calculating the effects of alpha bodies on beta bodies. And data block-out is efficient using the hierarchical mapping model.

4.4.3 Parallel Computation

The force interaction computation of many bodies is done by the back-end Data Parallel Units in the SIMD fashion. The following is a simple illustration to see how it is done in parallel:

1. The body mass, position and velocity data are distributed along the first row of the Maspar's square array of processors.
2. The body mass, position and velocity data are copied down the rows of the array. Then they are carefully copied across the columns, with the final result being that each processor will have a different set of pairs of bodies. .
3. The processors calculate in parallel all the accelerations for all the possible pairs of bodies they have.
4. This data is accumulated back up the columns to combine the interactions of all the pairs of bodies using recursive doubling technique.
5. Finally, the combined acceleration data is used to update the positions and velocities of all the bodies. How to update them depends on the simulation numerical method chosen. This computation happens in parallel for all the processors in the first row.

4.4.4 Simulation Numerical Method

As we have shown above, we first calculate the force interactions of bodies, we then get the acceleration of each body and we can update the positions and velocities of the bodies. Now, we need to consider how to compute the new position and velocity caused by the acceleration. We know that the change of position and velocity can be represented as:

```
delta_P = delta_t * V
delta_V = delta_t * A    in which,

V - velocity at time t
A - acceleration at time t
delta_t - time elapse from time t to t' (time-step)
delta_P - change of position from time t to t'
delta_P - change of velocity from time t to t'
```

These relations become more nearly exact as delta_t is diminished. They are differential equations. We need a numerical approximation method to solve those equations. The numerical method currently employed to simulate the bodies movement is called Euler-Heun Predictor-Corrector method with mop-up. It is a third-order Taylor method with the absolute error estimation as:

```
V_{true value} - V_{approx value} = Big-O (time-step^4)
```

The following are the iteration formulas of this method:

```
predictor:
    i = 0:
            Vp(i+1) = Vm(0) -- initial value
            Pp(i+1) = Pm(0) -- initial value
    i > 0:
            Vp(i+1) = Vm(i-1) + 2 * delta_t * Am(i)
            Pp(i+1) = Pm(i-1) + 2 * delta_t * Vm(i)
corrector:
    Vc(i+1) = Vm(i) + (delta_t / 2) * (Am(i) + Ap(i+1))
    Pc(i+1) = Pm(i) + (delta_t / 2) * (Vm(i) + Vp(i+1))
```

mop-up:

Vm(i+1) = (4 * Vc(i+1) + Vp(i+1)) / 5

Pm(i+1) = (4 * Pc(i+1) + Pp(i+1)) / 5

delta_t -	simulation time step
Pm(i-1) -	last iteration position mop up value
Vm(i-1) -	last iteration velocity mop up value
Pm(i) -	current iteration position mop up value
Vm(i) -	current iteration velocity mop up value
Am(i) -	current interaction mop up acceleration
Pp(i+1) -	next iteration position predicted value
Vp(i+1) -	next iteration velocity predicted value
Ap(i+1) -	next interaction predicted acceleration
Pc(i+1) -	next iteration position corrected value
Vc(i+1) -	next iteration velocity corrected value
Pm(i+1) -	next iteration position mop up value
Vm(i+1) -	next iteration velocity mop up value

The approximation accuracy depends on the time-step value as we have seen in the error estimation formula. It may not be accurate sometimes when we want to run the simulation in the interactive rate. We are still investigating new ways.

We also give the simple Euler's method as another approximation method so that we can compare these two methods' stability. The simple Euler's method is a first-order Taylor method, as following:

V(i+1) = V(i) + delta_t * A(i)

P(i+1) = P(i) + delta_t * V(i)

delta-t -	simulation time step
A(i) -	current interaction acceleration
P(i) -	current iteration position value
V(i) -	current iteration velocity value
P(i+1) -	next iteration position value
V(i+1) -	next iteration velocity value

4.4.5 Data Representation Conversion

Because of the different byte orders and different floating point formats of Pxpl5 host machine (Jason-SUN SPARC, using Big Endian and standard IEEE floating point representation) and Maspar front-end machine (Centurion-DEC VAX, using Little Endian and VAX-specific floating point representation), the DPU needs to convert the data format from its VAX formats to that used by Pxpl5 before blocking the data out. This conversion can be done in parallel.

4.5 Code Specifics

Here is a list of .c and .m files in this simulator:

psim.c - front-end control program.
fileio.c - routines for reading initial alpha bodies configuration.
convert.m - back-end routines for data representation conversion in parallel.
pc.m - many body simulation mpl program using predictor-corrector method.
euler.m - many body simulation mpl program using euler's method.

Next we list the major functions in this simulator:

main() - the simulation control program.
load_init_orb() - load initial setting of orbital bodies from user-specified file.
load_nbody_config() - load nbody data to the back-end processors.
load_orb_config() - load orbital bodies data to the back-end processors.
nbody_psim() - the back-end parallel nbody computation.
nbody_orb() - the back-end parallel orbital bodies computation.
send_nbody_posi() - send out updated positions and velocities to the front-end.
vax_to_sparc() - convert vax floating point representation to sparc format.
sparc_to_vax() - convert sparc floating point representation to vax format.

5. Appendicies

See the Pixel-Planes and head mounted display documentation included in the document.

6. Index