BBN Systems and Technologies
A Division of Bolt Beranek and Newman Inc.

BBN Report No. 7627

THE SIMNET

NETWORK AND PROTOCOLS

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1991	Technical Report

4. TITLE AND SUBTITLE

The SIMNET Network and Protocols

5. FUNDING NUMBERS

Contract Numbers:
MDA972-89-C-0060
MDA972-89-C-0061

6. AUTHOR(S)

Arthur R. Pope; Revised by Richard L. Schaffer

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Bolt Beranek and Newman, Inc. (BBN)
Systems and Technologies; Advanced Simulation
10 Moulton Street
Cambridge, MA 02138

8. PERFORMING ORGANIZATION REPORT NUMBER

BBN Report Number:
7627

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency (DARPA)
3701 North Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

DARPA Report Number:
None

11. SUPPLEMENTARY NOTES

None

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution Statement A: Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

Distribution Code:
A

13. ABSTRACT *(Maximum 200 words)*

A Simulation Network (SIMNET) project technical report describing the the SIMNET network and its communication facilities and protocols

14. SUBJECT TERMS

A technical description of the SIMNET network and its communication facilities and protocols.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Same as report.

Report No. 7627

The SIMNET Network and Protocols

Arthur R. Pope
Revised by Richard L. Schaffer

June 1991

Prepared by:

BBN Systems and Technologies
10 Moulton Street
Cambridge, Massachusetts 02138

Prepared for:

Defense Advanced Research Projects Agency (DARPA)
Information and Science Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

Preface

SIMNET: Advanced Technology for the Mastery of War Fighting

SIMNET is an advanced research project sponsored by the Defense Advanced Research Projects Agency (DARPA) in partnership with the United States Army. Currently in its sixth year, the goal of the program is to develop the technology to build a large–scale network of interactive combat simulators. This simulated battlefield will provide, for the first time, an opportunity for fully–manned platoon–, company–, and battalion–level units to fight force–on–force engagements against an opposing unit of similar composition. Furthermore, it does so in the context of a joint, combined arms environment with the complete range of command and control and combat service support elements essential to actual military operations. All of the elements that can affect the outcome of a battle are represented in this engagement, with victory likely to go to that unit which is able to plan, orchestrate, and execute their combined–arms battle operations better than their opponent. Whatever the outcome, combat units will benefit from this opportunity to practice collective, combined arms, joint war fighting skills at a fraction of the cost of an equivalent exercise in the field.

While simulators to date have been shown to be effective for training specific military skills, their high costs have made it impossible to buy enough simulators to fully train the force. Further, because of the absence of a technology to link them together, they have not been a factor in collective, combined arms, joint training. SIMNET addresses both of these problems by aiming its research at three high–payoff areas:

- Better and cheaper collective training for combined arms, joint war fighting skills.
- A test bed for doctrine and tactics development and assessment in a full combined arms joint setting.
- A "simulate before you build" development model.

These payoffs are achievable because of recent breakthroughs in several core technologies that have been applied to the SIMNET program:

- High speed microprocessors.
- Parallel and distributed multiprocessing.
- Local area and long haul networking.

- Hybrid depth buffer graphics.
- Special effects technology.
- Unique fabrication techniques.

These technologies, applied in the context of "selective fidelity" and "rapid prototyping" design philosophies, have enabled SIMNET development to proceed at an unprecedented pace, resulting in the fielding of the first production units at Fort Knox, Kentucky, just three years into the development cycle.

In addition to the basic training applications, work is underway to apply SIMNET technology in the area of combat development to aid in the definition and acquisition of weapon systems. This is made possible because of the low cost of the simulators, the ease with which they can be modified, and the ability to network them to test the employment of a proposed weapon system in the tactical context in which it will be used, i.e., within the context of the combined arms setting.

Work on SIMNET is being carried out by co–contractors Bolt Beranek and Newman, Inc. (BBN) and Perceptronics, Inc. Perceptronics is responsible for training analysis, overall system specification, and the physical simulators, and BBN is responsible for the data communication and computer–based distributed simulation and the computer image generation (CIG) subsystems. The project is a total team effort.

DARPA is the DoD agency chartered with advancing the state of the art in military technology by sponsoring innovative, high–risk/high–payoff research and development.

Table of Contents

1 INTRODUCTION

The SIMNET project has developed a network for simulating battles involving many vehicles by interconnecting large numbers of interactive vehicle simulators. This form of simulation is called distributed simulation because the computer systems supporting it—the individual vehicle simulators—may be distributed over local or large distances. These computer systems communicate by means of a network that allows them to exchange information quickly and efficiently. The manner of exchange is governed by a set of rules and conventions that we call the SIMNET protocols.

1.1 About this report

The purpose of this document is to describe the SIMNET network and its protocols at two levels. At one level, we have sought to provide a basic understanding of the mechanisms underlying a SIMNET distributed simulation. Beyond that, we have also attempted to document the protocols in sufficient detail to allow others to produce simulators that are fully compatible with the SIMNET system. Hence this report includes both a broad discussion of basic concepts and a somewhat more formal specification of the data communicated among simulators.

Someone desiring the basic understanding should examine chapters 1 through 4. The first two chapters cover some fundamental concepts underlying the distributed simulation. The third chapter focuses on an issue of critical importance to the distributed simulation: that of how, and how often, information about a vehicle's appearance in the simulated world must be communicated among simulators. A technique we employ to reduce the volume of communicated data, based on dead reckoning, is described in that chapter.

The fourth chapter specifies the characteristics required of the network supporting the simulation. Just as telephone conversations may be conveyed by a variety of media ranging from copper wire to light pulses, so too a distributed simulation can be supported by a variety of networks. The SIMNET protocols will operate on any network provided that network meets the requirements described in chapter 4. One network that meets these requirements—and with which we have experience—is the Ethernet™; appendix E describes the manner in which SIMNET protocols are made to use that network.

™ Ethernet is a registered trademark of the Xerox Corporation.

Of course, to support a distributed simulation a network must be capable of some minimum level of performance. What that level is depends on the size of the distributed simulation, and on the types of things being simulated. The methods and protocols described in this report are appropriate for distributed simulations both large and small, so we do not prescribe any particular level of network performance. As an example, however, chapter 4 describes our experience, which indicates that a network capable of carrying 1500 packets per second is required to support a distributed simulation involving 500 vehicles.

The remaining chapters of this report define the protocols in detail. They describe both the content of the data messages exchanged via the network, and the conditions under which these messages are produced. To allow the contents of the messages to be defined succinctly and unambiguously we employ a formal notation, which is described in appendix A. In chapter 5 we collect in one place the definitions of several basic data elements that appear in many of the messages. The messages themselves are defined in chapters 6 through 8.

The protocols employ a particular representational scheme for identifying the types of objects, such as vehicles and munitions. By this scheme, for example, an M1 tank is described as one type of object, and an M60 tank, as another. The scheme is intended to be extensible so that new types of objects can be incorporated into the framework established by the scheme without disrupting existing software implementations. Appendix B describes the overall scheme, and appendix C lists particular object types that have been defined within it.

Because the distributed simulation involves specific types of vehicle simulators performing detailed simulations of their vehicles, certain protocol messages must convey information that is vehicle–specific. We have defined the format of this vehicle–specific information for two types of vehicle simulators: the SIMNET M1 Abrams Main Battle Tank simulator [1], and the SIMNET M2/3 Bradley Fighting Vehicle simulator [2]. Appendix D defines the vehicle–specific aspects of the protocols for these simulators. Note that, although some portions of the protocol are permitted to be vehicle–specific, it is not necessary for all simulators to deal with these vehicle–specific portions.

This report supersedes three earlier reports describing the SIMNET network and protocols, [3], [4] and [17]. The protocols have undergone continued evolutionary development as part of the SIMNET project. Although further development is expected,

we believe that the current structure of the protocols ensures that the anticipated changes—such as the introduction of new types of vehicles—will have minimal impact on existing protocol implementations.

1.2 Distributed simulation

The simulated world about which we are concerned in this report is based on some region of terrain typically tens or hundreds of kilometers across. The terrain is populated with features such as hills, rivers, roads, trees, and buildings. Both the terrain and the features emplaced upon it are static: they do not change form in the course of a simulation.[1]

Operating on and above the terrain are vehicles that do change dynamically in the course of the simulation. Vehicles may move anywhere about the terrain, assume any orientation, and change appearance in any of a variety of ways. These vehicles are often simulated by interactive vehicle simulators operated at the direction of individuals or crews. A crew perceives the simulated world from the vantage point of their vehicle, wherever that happens to be. They see both the terrain around their vehicle, and the other nearby vehicles being operated by other crews. Events unfold in this simulated world at a pace which is simply that of real time.

Because the simulated world exists as a place where battles are conducted, certain other phenomena are also found there. These phenomena include:

- Weapons fire and the effects it has upon vehicles.
- Supplies of fuel and ammunition, and the transfer of these supplies among vehicles.
- Vehicle malfunctions and repairs to correct these malfunctions.
- Radar emissions and detection by radar.

We use the term *exercise* to refer to a simulation conducted over some period of time involving some simulated world. The computer systems that simulate this world we call *simulators*. A number of simulators may participate in an exercise at one time and they

---

1 It is not that the protocols preclude changes to the terrain; they simply make no provisions, in their present form, for representing and distributing information about terrain changes.

must, of course, share information about the world they are simulating. This information includes:[2]

- Data required by a simulator in order for it to begin participating in an exercise.
- Descriptions of the locations and appearances of vehicles.
- Descriptions of events related to weapons fire and collisions.
- Reports of the exchange of fuel or ammunition among vehicles.
- Descriptions of repairs completed on vehicles.

Simulators share information by means of a network that interconnects them. The network may span short distances (a local area network) or large ones (a long haul network), or it may be some combination of both local area and long haul networks. Although the network must meet certain basic requirements, its exact topology is not important to a discussion of the protocols. All that is required of the network topology is that each simulator attach to it at some point.

The exchanges among simulators are governed by a set of protocols that have been designed with several goals in mind. Among these goals are:

- The protocols must ensure that a sufficiently consistent model of the simulated world is shared among all simulators.
- The protocols must allow simulators to begin and end their participation in an exercise at any time without disruption to the exercise.
- The protocols must be extensible in future to accommodate new types of vehicles, weapons, and other simulated phenomena without requiring significant changes to existing simulator implementations.
- The protocols must minimize the amount of information to be exchanged via the network, thereby minimizing the requirement for network throughput.
- The protocols must allow an optimal balance to be achieved between the amount of computation performed by simulators, and the amount of information that must be exchanged among them.

---

2 Note that information about the terrain is not among the things simulators exchange via the network in the course of a simulation. Each simulator is assumed to have access to a description of the terrain.

- The protocols must allow the computing tasks necessary for modeling phenomena in the simulated world to be distributed appropriately according to where these tasks may best be performed.
- The protocols must provide sufficient information about events in the simulated world to support later reconstruction and analysis of those events.

A distributed simulation can encompass many different types of simulators. Two quite different types of simulators that have been developed as part of the SIMNET project serve to illustrate the range of possibilities. The M1 tank simulator is operated by a full crew of four who control their single, simulated tank much as they would an actual tank. The SIMNET Semi–Automated Forces system [5] on the other hand, allows a few individuals to direct a large number of ground and air vehicles that operate as a unit in the simulated world. One simulates a single vehicle; the other simulates many. Simulators of both types can cooperate together in a single, distributed simulation. In this report, wherever we need to distinguish a simulator as being one that simulates a single vehicle at the direction of a full crew, we will call it a *crewed vehicle simulator*.

One other type of simulator we refer to in this report is the SIMNET Management, Command, and Control (MCC) system [6]. As a simulator, it simulates a variety of combat support and combat service support vehicles under the direction of a few individuals, but it also plays an administrative role by initiating other, crewed vehicle simulators into the distributed simulation. This combination of multiple functions in a single system is not in any way a requirement of the SIMNET protocols, but it serves as a further example of how the protocols will accommodate various types of simulation systems.

Most of the messages exchanged via the network are multicast so that they can be received by any system on the network. This makes possible systems that, by listening on the network, can report or record all events happening in the simulated world. One such system, which we call a *Data Logger*, simply records messages as they appear on the network, noting the time of each one. The record produced by the Data Logger can be used to analyze, review, or even replay all or part of an exercise.

Although the principle purpose of the network is to convey information about the simulated world for use by simulators, it has other purposes as well. The network is used by simulators to report supplemental information that may be useful in certain analyses of an exercise. This information includes, for example, the status of a simulated vehicle's internal subsystems and stores of supplies. Computers collecting this supplemental

information from the network can aid analysts in interpreting the events taking place in the simulated world.

1.3 Scope of this work

Three points must be made concerning the scope of the work described in this report. First, this report addresses the problem of linking together simulators—with each simulator modeling one or more vehicles—so that a large collection of simulated vehicles can interact in a simulated world. This problem is to be distinguished from that of linking multiple computer systems together to create a single vehicle simulator. The two problems are quite different in character, and they may demand different solutions.

Second, the techniques we have specified for linking together vehicle simulators are meant to build upon, rather than replace, standard communication services. Our intent is not to recommend a particular choice of network service, but rather to describe how a network meeting certain requirements can be employed for distributed simulation. To date we have used Ethernet, a standard, local–area–networking technology. Other network services may prove to be as appropriate or better. Alternatives include the Fiber Distributed Data Interface (FDDI) local area network; the DoD Internet Protocol (IP) and ISO connectionless internetwork protocol extended to support multicasting; and the DARPA internet stream protocol (ST) [7]. The distributed simulation protocols described in this report could be carried by any of these.

There is another aspect to the distinction we draw between the content of information communicated, and the mechanism by which it is carried. We have linked together local area networks of simulators using long haul networks and gateways so that simulators at various sites may participate in a common exercise. The simulators themselves use the same protocols regardless of whether they are interacting locally or over a long haul network. The gateways allow the long haul network to be included in a manner that makes it transparent to the simulators, and requires no change to the protocols used among simulators. We are continuing our study of gateway–to–gateway protocols that will provide efficient utilization of a standard internet—such as one employing IP or ST protocols—while supporting distributed simulation. The third point to be made about the scope of this report, therefore, is that it does not describe gateway–to–gateway protocols. However, the simulator–to–simulator protocols it does describe are identically effective regardless of whether the simulators that use them are together on a single local area network, or separated onto distinct local area networks that are linked by gateways and a

long haul network. Our chief aims in developing a gateway–to–gateway protocol are to retain this sense of network transparency, and to retain the simulator–to–simulator protocols in their present form.

2 DISTRIBUTED SIMULATION CONCEPTS

This chapter introduces concepts that provide a framework for the definition of distributed simulation protocols.

2.1 Architecture

Distributed simulation operates in a particular network environment called a *distributed simulation internet.* This network environment may consist of a single local area network, or it may include a series of local area networks linked together by a long haul network. Local area networks are referred to as *sites*, and the computers at each site (attached to a local area network) are referred to as *simulators*. This arrangement is illustrated in figure 2–1.

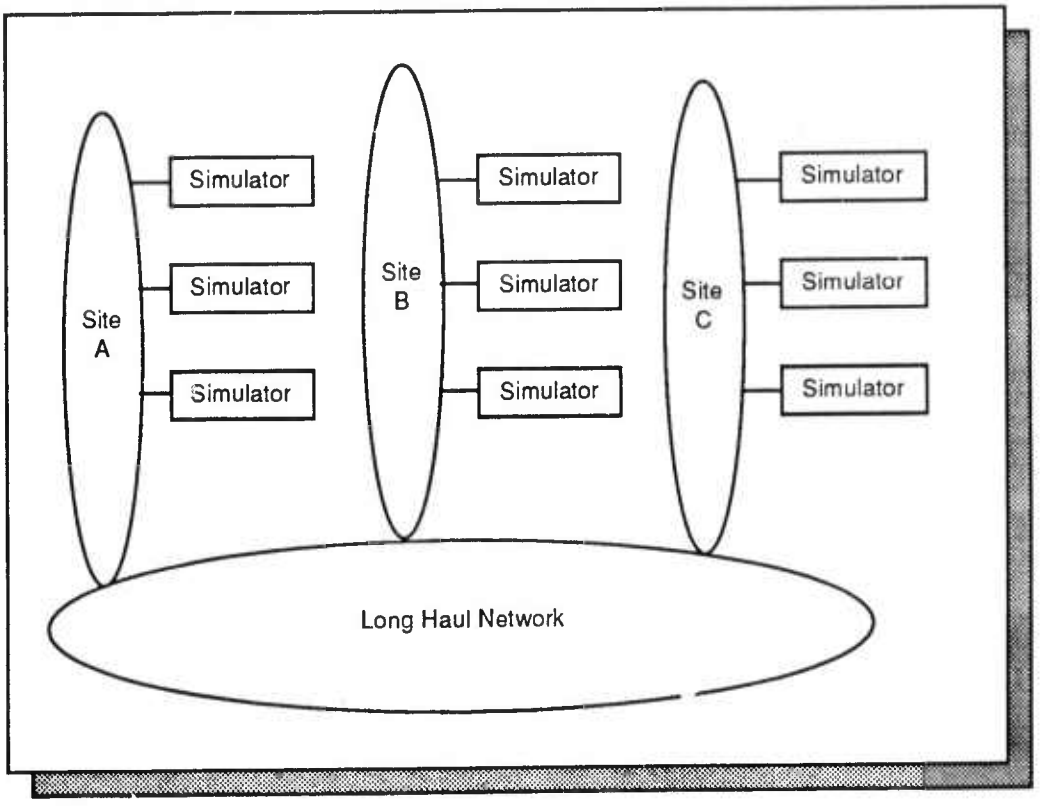


Figure 2–1. A distributed simulation spans a collection of simulators located at various sites, connected by local area and long haul networks. The overall network environment is called a distributed simulation internet.

Typically, each simulator may be engaged in some aspect of an overall distributed simulation; it may, for example, be simulating one vehicle. However, for convenience,

the term *simulator* is applied to all computers participating in the distributed simulation, including those that are only "listening" to a simulation exercise passively rather than simulating anything

A distributed simulation is implemented using a family of related protocols, each serving a particular need of the distributed system. These protocols include:

- A *simulation protocol*, used to introduce simulated elements into an exercise, remove them from an exercise, and convey information about the simulated world for use by simulators.
- A *data collection protocol*, used to report information arising from the simulation that is (a) of interest primarily to those studying the course of an exercise, or (b) needed to restart an exercise following an interruption.
- An *association protocol* providing some communication services that are both particular to the application of distributed simulation, and needed to support the simulation and data collection protocols.

Simulators engaged in a distributed simulation implement appropriate features from each of these protocols, and participate in all of them simultaneously. For example, a simulator that is involved in an exercise will be reporting information about its behavior to other simulators using the simulation protocol, and reporting data for collection and analysis using the data collection protocol Messages of both the simulation protocol and the data collection protocol will be conveyed using the association protocol. All three protocols are described in detail in this report.

The protocols, in turn, are based on the use of a communication service that may be implemented in various ways. We describe in this report how one communication network, Ethernet, may be used to provide this service. Figure 2–2 shows how the three protocols and the underlying communication service are related to each other.

Simulators may engage in other communication protocols besides those used to achieve distributed simulation. A simulator might implement additional protocols for functions such as remote diagnosis or bulk transfer of data, and use the same underlying communication service to support these additional protocols. This report, however, is concerned only with the three protocols that provide distributed simulation.

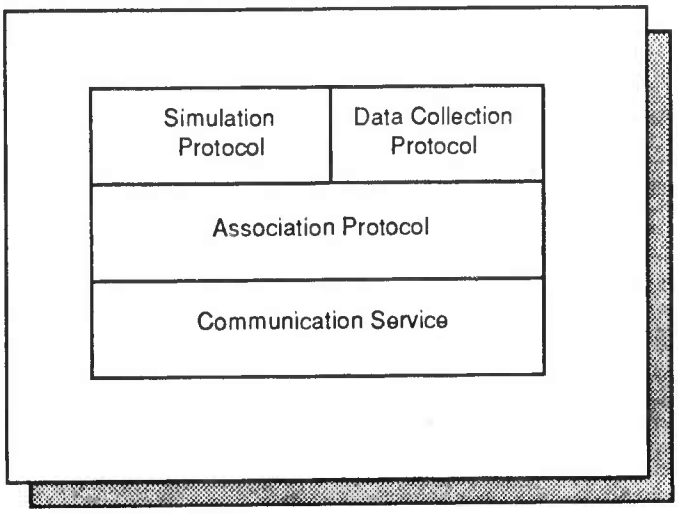


Figure 2–2. The simulation and data collection protocols are carried by the association protocol, which, in turn, is supported by an underlying communication service.

The simulation protocol and the data collection protocol share many aspects of data representation and style. Moreover, there are cases where the interactions of one protocol are closely related to those of the other. For example, when a simulator joins an exercise through a simulation protocol interaction, it begins to make data available through data collection protocol interactions. The division of functions among these two protocols is intended to provide a logical and convenient grouping of those functions rather than a distinct separation of them.

The information exchanged by computers as part of a protocol is packaged in messages called *protocol data units (PDUs).* For each of the SIMNET protocols, a particular set of PDU types is defined according to the communication needs of that protocol.

2.2 Simulation exercises

A simulation exercise is a joint activity in which multiple simulators share a common, simulated world. Associated with any exercise are certain things that must be known to each participant. These things include:

- *Information about the terrain upon which the exercise is taking place.* Each simulator is assumed to have access to any information it requires about the terrain. Of course, simulators must agree closely on how the terrain is shaped, and how it is covered with features such as vegetation and buildings. The SIMNET protocols impose no constraints on how terrain information is represented or used by simulators. The protocols do, however, provide a mechanism for identifying the terrain information to be used for a particular exercise. A collection of information describing a particular area of terrain is referred to as a *terrain database*; each terrain database is identified by a combination of name and version number.
- *The date and time in the exercise's simulated world.* Although time in the simulated world passes at exactly the same pace as it does in the real world, clocks may be set differently there. The SIMNET protocols convey the value of time in the simulated world so that simulators can vary effects such as lighting and visibility according to the value of simulated date and time of day.
- *The identity of the exercise.*. The SIMNET protocols allow multiple exercises to occur simultaneously using a single network while treating each as though it were being supported by its own network. The concurrent exercises are kept from interfering with each other by the assignment to each exercise of a distinct integer called an *exercise identifier*. All PDUs pertaining to a particular exercise bear that exercise's identifier when transmitted over the common network. The recipient of a PDU simply ignores the PDU if it bears the identifier of an exercise other than the one in which it is currently participating.

The SIMNET protocols provide two mechanisms for distributing this information among exercise participants. Using a feature of the data collection protocol, any simulator may query for and obtain this information from other participants of an exercise. The simulation protocol contains a second mechanism that allows one simulator to initiate another into an exercise while providing it with the information. This latter process is called *activation*. Either or both mechanisms may be used. It is nevertheless required that a proper exercise identifier, terrain database, and simulated time be chosen prior to an exercise, and that this information first be supplied to some participating simulator through a means not encompassed by the SIMNET protocols.

2.3 Simulated vehicles

The SIMNET protocols are intended to accommodate a broad variety of different types of vehicle simulators. An individual simulator participating in an exercise may model a single vehicle—as does the M1 tank simulator—or many vehicles—as does the MCC system. A single vehicle may be controlled by a full complement of human crew

members, or many vehicles may be controlled by a single person. A vehicle may be either manned, such as a tank or aircraft, or unmanned, such as a missile.

A simulator may begin to involve its vehicle in an exercise at any time, provided that the vehicle joining the exercise has correct values for such parameters as the exercise identifier. The vehicle can be introduced into the exercise by the simulator itself, with parameters provided by its human operators. Alternatively, a simulator's vehicle can be introduced into an exercise by another computer, such as an MCC system, through the process of activation.

Once a vehicle is involved in a simulation, it is said to be *active*. At any time that vehicle's simulator can terminate its involvement in the exercise while announcing the vehicle's withdrawal to other simulators. Alternatively, the vehicle can be removed from the exercise by another computer—such as the MCC system that activated it. In either case, the process is called *deactivation*, and it is conducted via the simulation protocol.

Every vehicle participating in an exercise has assigned to it a distinct number called a *vehicle identifier*. A system that simulates many vehicles must have a unique vehicle identifier for each one, and a system that activates other simulators must provide them with appropriate vehicle identifiers to use. No two vehicles in the same exercise may have the same vehicle identifier.

In addition to a vehicle identifier, each vehicle participating in an exercise has several attributes that its simulator makes known to all others via the simulation and data collection protocols. These attributes include:

- *Which side the vehicle is fighting for.* The vehicles participating in an exercise are grouped into collections we call *forces*. Typically, two forces are involved and these forces fight against each other. However, the protocols allow vehicles to be divided among many different forces, and they impose no restrictions as to which forces fight with or against which others. Forces are identified by numbers in the range 1 through 255.
- *What organizational unit the vehicle is allocated to.* Within a force, vehicles are allocated among various organizational units that are arranged in some hierarchy. For example, a vehicle may belong to a certain company, of a certain battalion, of a certain brigade, etc. The protocols provide a way for advertising a vehicle's position within its force's organizational hierarchy, for any of various forms of military hierarchy.

- *What type of vehicle it is.* A vehicle's type identifies it as a particular kind of vehicle, such as, for example, an M1A1 Abrams main battle tank or a Soviet HIND–E attack helicopter. There are three vehicle types associated with each vehicle: one is the type of vehicle that is actually being simulated; the other two are called the vehicle's *guises*, and they define how the vehicle appears to other observers. Often, all three vehicle types are the same with the result that the vehicle appears identically to all observers. The vehicle types can be made to differ, however, to obtain a useful effect.

  Each vehicle has two guises so that it can be made to appear as one type of vehicle to those of one force, and as another type of vehicle to those of other forces. Based on whether an observer's vehicle belongs to a particular force (force number 1) that observer's simulator will display other vehicles using one guise or the other. One application of this feature is to support a battle between two forces, each of which views themselves as using U.S.–type vehicles, and the other force as using Soviet–type vehicles. All vehicles in the battle may be simulated as M1 tanks, but those of force 1 may be disguised as T72 tanks to those of force 2, while those of force 2 are disguised as T72 tanks to those of force 1.

- *Where the vehicle is, and how it is oriented.* A vehicle need not always be visible during an exercise—some vehicles may vanish from one place to later reappear at another. At all times when a vehicle is active and visible, however, it has a location and orientation in space. These are described in terms of a world coordinate system discussed in section 2.4. Some vehicles have independently movable parts, such as a turret and a gun barrel, whose relative positions are also described.

- *An optional vehicle marking.* A vehicle may be seen by its observers as bearing a label such as a name (e.g., "Titanic") or a bumper number (e.g., "PltLdr/3/C"). Whether and how the label is displayed for a particular observer may depend on which forces the vehicle and its observer belong to.

- *Variations on the basic appearance of the vehicle.* A vehicle's basic appearance can be modified in various ways. For example, it can catch fire, emit a plume of smoke, or become destroyed. Some variations, such as those just listed, are applicable to almost all types of vehicles; other variations, such as whether an M2 infantry vehicle's rear ramp is lowered, apply only to a specific type of vehicle.

- *The vehicle's engine speed.* A vehicle's engine speed is reported via the simulation protocol to make possible a simulation of sounds on the battlefield.

- *What the vehicle is capable of.* One vehicle may be called upon by another to supply munitions or perform repairs. Whether a vehicle is capable of providing these services is indicated via the simulation protocol.
- *How various subsystems of the vehicle are operating.* A vehicle's simulator may model the vehicle to a level of detail where the operational status of various vehicle subsystems are represented. The simulation and data collection protocols provide mechanisms for reporting the status of a vehicle's subsystems, and for flagging changes in subsystem status.
- *What munitions, such as fuel and ammunition, the vehicle is carrying.* Simulators that model the quantities of stores of various kinds carried by their vehicles report this information via the data collection protocol.

A vehicle's identifier, type, and force assignment are attributes that do not change in the course of an exercise. Other attributes change dynamically in a manner that requires the vehicle's simulator to periodically inform other simulators of the changes.

A vehicle's appearance to observers is determined by attributes such as its guises, location, and orientation. Whenever a vehicle's appearance changes in any significant way, that vehicle's simulator must inform other simulators of the vehicle's new appearance. The simulator does this by issuing an update message—called a *Vehicle Appearance PDU*—to all of the other simulators participating in the exercise. In chapter 3 we describe these update messages, the conditions that compel simulators to send them, and the behavior required of simulators receiving them.

2.4 Coordinate systems

Locations in the simulated world are identified using a right–handed Cartesian coordinate system called the *world coordinate system*. The axes of this system are labelled X, Y, and Z, with the positive X axis pointing east, the positive Y axis pointing north, and the positive Z axis pointing up. A distance of one unit measured in world coordinates corresponds to a distance of one meter in the simulated world, and a straight line in the world coordinate system is a straight line in the simulated world.

Since simulators express locations to each other in terms of the world coordinate system, all must share a common definition of where the origin of that coordinate system lies. Moreover, in order to maximize the precision with which locations can be expressed, the origin should be chosen so that the space used by simulated vehicles lies relatively near

the origin. These are the only constraints that the SIMNET protocols impose on the origin of the world coordinate system. By convention, however, the origin is usually placed at the southwest corner of the terrain area.

To describe the location and orientation in space of any particular vehicle, we introduce a *vehicle coordinate system* that is fixed to that vehicle. This is also a right–handed Cartesian coordinate system with meter–sized units; its X axis points to the vehicle's right, its Y axis points to the vehicle's front, and its Z axis points up. A convention is used for the position of the coordinate system's origin: the origin of a ground vehicle's coordinate system is at the center of the vehicle's base; that of an air vehicle is in the middle of its fuselage.

The location of a vehicle is specified as the position of the origin of its vehicle coordinate system, expressed in world coordinates. The orientation of a vehicle is specified as the relative rotation between its coordinate system and the world coordinate system. In the context of the SIMNET protocols, this rotation is represented as a nine element rotation matrix:[3]

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

---

[3] The rotation matrix has several equivalent interpretations. When the world and vehicle coordinate systems share a common origin, the following are all true:

- The three columns of the matrix correspond to unit vectors lying along each of the three positive axes of the vehicle coordinate system, expressed in world coordinates.
- The three rows of the matrix correspond to unit vectors lying along each of the three positive axes of the world coordinate system, expressed in vehicle coordinates.
- When a vector expressed in vehicle coordinates is premultiplied by the matrix, the result is the same vector expressed in world coordinates.
- When a vector expressed in world coordinates is postmultiplied by the matrix, the result is the same vector expressed in vehicle coordinates.

2.5 Events

In the course of an exercise, various PDUs are issued by simulators to announce certain events involving the vehicles they simulate. These PDUs report such occurrences as collisions between vehicles, shots fired from vehicles, shells striking their targets, and injuries suffered by vehicles. In studying an exercise it is often desirable to be able to link these events, establishing associations between causes and effects. Given a PDU describing the firing of a shell, for example, it must be possible for the analyst to locate a later PDU describing the explosion of that shell, and any further PDUs describing how other vehicles were damaged by the explosion.

To make these associations explicit, the various PDUs that describe a related series of events are linked with each other by virtue of their bearing a common tag. The tag is created by the simulator whose vehicle initiated the chain of related events, perhaps by firing at or colliding with another vehicle. The tag consists of a pair of identifiers provided by that simulator: its vehicle's identifier, and a unique serial number, called an *event identifier*, generated by the simulator. Each time a vehicle initiates a new chain of events, its simulator creates a new, unique event identifier on behalf of that vehicle. From the time a vehicle enters an exercise until the time it withdraws, each event identifier created for the vehicle must be a new, unique one. Since the event identifier is unique among those created for the vehicle, and since the vehicle's own identifier is unique among all vehicles in the exercise, the combination of the two identifiers uniquely labels the chain of events.

This is how the pair of vehicle and event identifiers serves to link the PDUs that report the events of a weapons engagement. When a vehicle fires, its simulator generates a new event identifier and issues a PDU containing both that event identifier and its own vehicle identifier. When the fired round impacts, the simulator issues a second PDU bearing the same pair of identifiers. Then any simulator whose vehicle is damaged by the exploding round reports its damage by issuing a PDU that contains the same pair of vehicle and event identifiers. These PDUs, and the manner in which they convey vehicle and event identifiers, are described in chapters 7 and 8.

The sequences of PDUs that describe a collision between vehicles, or a repair to a vehicle, are also tied together by the PDUs sharing a common, unique pair of vehicle and event identifiers.

3 VEHICLE APPEARANCE

Much of the information that must be communicated among simulators participating in a distributed simulation is that which describes the appearance of vehicles as they move about the simulated world. In this chapter, we describe the method used in the simulation protocol for communicating vehicle appearance information.

3.1 Overview

As a simulator models the behavior of a vehicle in real time, that vehicle's appearance can be constantly changing. The vehicle may be changing its orientation and location, moving its turret or gun barrel, and even catching fire and burning. The vehicle's simulator must inform other simulators of these changes so that all simulators participating in the exercise can depict the vehicle correctly, at its current location.

The appearance of a vehicle is completely described by a Vehicle Appearance PDU, which is defined as part of the simulation protocol. This PDU identifies a vehicle and describes that vehicle's type, location, and orientation. The PDU also describes whether the vehicle is on fire, destroyed, or emitting a plume of smoke. If the vehicle has independently movable parts, such as a turret and gun barrel, the PDU describes the relative positions of those parts. Finally, for reasons we will explain, the Vehicle Appearance PDU may contain information about the vehicle's motion, such as its velocity vector.

It would be possible for the simulator of a vehicle to issue a Vehicle Appearance PDU describing that vehicle every single time the vehicle's appearance changed. However, while the vehicle was in motion, PDUs would be issued as frequently as the simulator recomputed the location of the vehicle, which could be quite often.

The simulation protocol allows us to reduce the frequency with which Vehicle Appearance PDUs must be issued by employing a technique called *dead reckoning*. The term, borrowed from navigation, means establishing the position of a ship based on an earlier known position and estimates of time and motion. Simulators may use dead reckoning to extrapolate the locations of vehicles so that they need obtain less often the actual Vehicle Appearance PDUs describing those vehicles.

This is how dead reckoning is used. Each simulator is responsible for maintaining a detailed model of its own vehicle's state, including, for example, engine power, thrust, and fuel consumption; aerodynamic forces or terrain forces; weapon systems computers, etc. The simulator will have a precise notion of its own vehicle's appearance over time. Each simulator also maintains a simple dead reckoning model of the state of all other vehicles—simulated by systems elsewhere on the network—with which it might possibly interact. Typically, these are all the other vehicles within a particular range of the simulator's own vehicle. The dead reckoning model is maintained by extrapolating the last reported location of each other vehicle, based on its last reported velocity vector, until such time as a new Vehicle Appearance PDU is received.

This approach implies that each simulator is also responsible for issuing a new Vehicle Appearance PDU whenever its vehicle changes course or speed. To do this, each simulator must maintain, in addition to its "high fidelity" model, a dead reckoning model that corresponds to the model that other simulators are maintaining of its vehicle. After each update of both its high fidelity model and its dead reckoning model, the simulator compares the exact appearance of its vehicle with the extrapolated appearance and issues a Vehicle Appearance PDU only when a significant discrepancy has accumulated.

This approach obviously leads to a variable rate of issuing Vehicle Appearance PDUs that will differ from one simulator to another at any given time. Each simulator transmits these PDUs only when necessary. The principal motivation is, of course, to minimize network communication traffic and hence the amount of incoming information that each simulator must process.

In essence, dead reckoning achieves a trade off among three factors: the network communication traffic, the amount of computation performed by simulators, and the precision with which each simulator perceives the vehicles of other simulators. Network traffic is reduced by dead reckoning because fewer Vehicle Appearance PDUs are transmitted. Computation demands are increased for the simulators that must, as a result, extrapolate the appearances of vehicles in the absence of any Vehicle Appearance PDUs describing them. And precision is limited by the amount of discrepancy allowed to accumulate between a vehicle's high fidelity model and its dead reckoning model.

There are many parameters of the dead reckoning algorithm that may be adjusted to establish the point at which these three factors are balanced. The thresholds against which discrepancies are gauged must be carefully chosen, for as these thresholds are

increased network traffic is reduced, but so is precision. There are also choices to be made among dead reckoning algorithms. Dead reckoning can be based on the use of higher order time derivatives of vehicle motion—such as acceleration—with the result that network traffic is reduced, but more computations must be performed to use these higher order derivatives.

The optimal choice of discrepancy thresholds and dead reckoning algorithms depends on the type of vehicle simulated. The choices that are appropriate for slow moving ground vehicles may not be optimal for high–speed aircraft. Thus the simulation protocol allows different thresholds and algorithms to be used for different types of vehicles, and vehicles are classified according to the method used for dead reckoning them. These classifications are called *vehicle classes*. Currently, three classes are defined:

static	Vehicles of this class are always stationary while visible, and they have no independently moving parts. The location of one of these vehicles need not be updated through dead reckoning because its velocity is always zero.
simple	Vehicles of this class may move, but they have no independently moving parts. The location of one of these vehicles is dead reckoned using its velocity vector.
tank	Vehicles of this class may move, and they have independently moving turrets and gun barrels. The M1 tank and the M2 fighting vehicle are both in this class. The location of one of these vehicles is dead reckoned using its velocity vector.

3.2 Measuring discrepancies in vehicle appearance

Before defining the discrepancy thresholds and dead reckoning algorithms used for vehicles of each of these classes, we first describe some discrepancy measures that are appropriate to all vehicles. These are measures of the difference between a vehicle's appearance as determined by its high fidelity model and that determined by its dead reckoning model.

3.2.1 Discrete appearance attributes

A vehicle's appearance is partly described by a series of discrete attributes indicating such things as whether the vehicle appears destroyed, whether it is on fire, and whether it is emitting a plume of smoke. The discrepancy between two versions of a vehicle's appearance, therefore, includes a measure of how these discrete attributes differ among the two versions.

3.2.2 Location

A vehicle's location is defined as the location of its vehicle coordinate system's origin, in world coordinates. The discrepancy between two versions of a vehicle's location, however, is measured in the coordinate system of that vehicle. Specifically, we measure the discrepancy between the actual location of a vehicle and its location as predicted by dead reckoning, simply as the dead reckoned location transformed into the vehicle's actual coordinate system. This is a three–element vector that describes the discrepancy along each of the vehicle's three axes. By using the vehicle's own coordinate system for this measure, we can apply different thresholds to discrepancies along each of the three vehicle coordinate system axes. We are thus able to have different tolerances for longitudinal than for lateral departures between actual and dead reckoned locations of the vehicle.

3.2.3 Orientation

A vehicle's orientation is represented by a nine element rotation matrix. The discrepancy between two versions of a vehicle's orientation can also be represented by such a matrix, describing the relative rotation between the two vehicle orientations. This relative rotation matrix may be obtained by multiplying the vehicle's actual rotation matrix by the transpose of its dead reckoned rotation matrix. If R is a relative rotation matrix obtained in this way,

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = (R_{actual})\ ((R_{dead\ reckoned})^T)$$

then we can derive from R a simple measure of the amount of rotation that matrix represents. This measure is the rotation expressed as an angle measured about a single, arbitrary axis. The direction of the axis is not significant for our application, but we use the magnitude of the rotation as a measure of discrepancy. The magnitude of rotation is represented by θ in the equation

$$\cos\theta = \frac{r_{11} + r_{22} + r_{33} - 1}{2}$$

This scalar angle of rotation about an arbitrary axis is the measure we use to describe the discrepancy between a vehicle's actual orientation, and its dead reckoned orientation.

3.3 Dead reckoning methods and discrepancy thresholds

We now define the discrepancy thresholds and dead reckoning methods applied to the various classes of vehicles.

3.3.1 Vehicles of the static class

A vehicle of the static class is always stationary while visible. Therefore, no dead reckoning is performed of either its location or its orientation. The simulator of a static class vehicle must issue a new Vehicle Appearance PDU describing the vehicle whenever one of its discrete appearance attributes changes.

3.3.2 Vehicles of the simple class

A vehicle of the simple class is modeled using dead reckoning based on the vehicle's velocity vector. This vector is computed by the vehicle's simulator and included in any Vehicle Appearance PDUs that are issued to describe that vehicle.

The simulator of a simple class vehicle must issue a new Vehicle Appearance PDU whenever any of the following discrepancies accumulate between the appearance of the vehicle as determined by its high fidelity model, and its appearance as determined by dead reckoning:

- A difference in any of the vehicle's discrete appearance attributes.
- A difference in location, as measured along any of the three vehicle axes, that is greater than 10% of the vehicle's dimension along that axis.
- A difference in orientation that is greater than 3 degrees.

Other values of the location and orientation thresholds may be chosen to meet the needs of a particular training system or exercise. This report does not address the question of how the location and orientation threshold values are provided to simulators.

3.3.3 Vehicles of the tank class

A vehicle of the tank class is modeled using dead reckoning based on the vehicle's velocity vector. This vector is computed by the vehicle's simulator and included in any Vehicle Appearance PDUs that are issued to describe that vehicle. Because the vehicle

has an independently movable turret and gun, the Vehicle Appearance PDUs also contain the relative positions of those parts.

The simulator of a tank class vehicle must issue a new Vehicle Appearance PDU whenever any of the following discrepancies accumulate between the appearance of the vehicle as determined by its high fidelity model, and its appearance as determined by dead reckoning:

- A difference in any of the vehicle's discrete appearance attributes.
- A difference in location, as measured along any of the three vehicle axes, that is greater than 10% of the vehicle's dimension along that axis.
- A difference in orientation that is greater than 3 degrees.
- A difference in turret azimuth, as measured relative to the vehicle's hull, that is greater than 3 degrees.
- A difference in gun elevation, as measured relative to the vehicle's hull, that is greater than 3 degrees.

Other values of the location, orientation, azimuth, and elevation thresholds may be chosen to meet the needs of a particular training system or exercise. This report does not address the question of how the values of the thresholds are provided to simulators.

3.4 The effect of delay

A description of a vehicle's appearance passes through several hands from the time it is first expressed as a Vehicle Appearance PDU, to the time the vehicle is displayed by an observer's simulator. The steps include processing by the software that provides communication services, transmission across a network, and perhaps queuing within the receiving simulator. Each of these steps may impose some delay, with the result that the observer is always seeing the vehicle as it was at some point in the recent past.

When a Vehicle Appearance PDU incurs a delay in travelling from its sender to an observer, the observer perceives the sending simulator's vehicle not where it is at that moment, but where it was when the PDU was sent. The magnitude of this discrepancy is proportional to the speed of the vehicle described by the PDU, and to the magnitude of the network delay. Therefore, this effect is expected to be most evident in certain situations—such as when aircraft flying at high speed are able to observe each other

closely (e.g., they are in close formation) while being simulated by widely separated simulators.

Nevertheless, in many distributed simulation situations a constant delay of even half a second is not apparent. Using a network composed of Ethernets and terrestrial long–haul circuits, we have found no evidence that the delay is a problem. More easily noticed, however, are the effects of any variance in the delay from one Vehicle Appearance PDU to the next. If the delay between sender and observer varies, then the motion of the vehicle may not appear smooth. In this case, the magnitude of the effect is proportional to the speed of the vehicle, and to the variance of the delay. The effect may show up, for example, when an air defense vehicle is tracking a fast–moving aircraft. If there exists variance in the delay with which the air defense vehicle simulator displays its image of the aircraft, and if the variance is not somehow compensated for, then the aircraft will be seen to "jump" forward and backward as new Vehicle Appearance PDUs are received.

The simulation protocol includes a mechanism by which delay variance can be compensated for. The mechanism requires some extra computation by an observer's simulator, but—at the discretion of that simulator—this computation need not be performed for all vehicles. An observer's simulator might choose to perform the extra computation, for example, only for fast–moving aircraft that are near enough to be tracked closely.

The mechanism uses timestamps that are reported at a millisecond scale. Each simulator maintains its own millisecond clock, which need not be synchronized with that of any other simulator. When issuing a Vehicle Appearance PDU, a simulator includes the current value of its clock. A simulator receiving consecutive Vehicle Appearance PDUs describing a single vehicle can examine the timestamps in those PDUs to measure, and compensate for, any delay variance.

Here we prescribe one method a simulator might use to compensate for delay variance of the Vehicle Appearance PDUs it is receiving.

- Upon receiving a Vehicle Appearance PDU describing a particular vehicle for the first time, the simulator simply records the information contained in that PDU, including the timestamp. It uses this information to display the vehicle.

- Each time the simulator must use dead reckoning to extrapolate the appearance of the vehicle, it increments its record of the vehicle's timestamp by the period of time over which it is extrapolating. It then uses the updated appearance information to display the vehicle.
- When a subsequent Vehicle Appearance PDU describing the same vehicle is received, the simulator compares the timestamp in the PDU with that it has recorded (and updated while dead reckoning). If the two are in close agreement, then the contents of the new PDU are adopted immediately. Otherwise, the simulator either (a) extrapolates forward the appearance described in the new PDU to the time of its recorded timestamp, or (b) sets aside the PDU until its recorded timestamp matches that in the PDU.

The SIMNET protocols do not mandate this particular method of compensating for delay variance, or even require the use of any delay variance compensation whatsoever. Whether and how compensation is performed by any particular simulator will depend on the nature of the network supporting the distributed simulation, and the use to which the simulator will be put. The protocols do require, however, that all Vehicle Appearance PDUs contain the timestamp that permits a method such as this to be employed by any receiving simulator.

4 THE SIMNET NETWORK

The SIMNET protocols are supported by a network whose capabilities are defined in this chapter.

In terms of the ISO Basic Reference Model for Open Systems Interconnection [8], the SIMNET protocols are Application Layer protocols that make direct use of network or data link layer services with no specific services required of the intermediate layers.

4.1 Network requirements

Any network will support a distributed simulation using the SIMNET protocols provided it meets the criteria listed below. These criteria apply regardless of whether the network is a single, local area network, or a large distributed simulation internet composed of many subnets.

- The network must provide for *connectionless data transfer*, also known as a *datagram service*. This means that a computer on the network must be able to to transfer data to another computer on the network in a single operation, without first establishing a connection with the destination computer. The unit of data transferred in a single operation is called a *datagram*.
- A datagram must be able to convey at least 2048 bits (256 octets) of information.
- The network must provide either for the *broadcasting* of datagrams, or for the *multicasting* of datagrams. A broadcast datagram will be delivered to all computers on the network (other than the sender). A multicast datagram will be delivered to a subset of all computers on the network. (Broadcasting is actually a special case of multicasting.)
- The network should have a low rate of non–delivery. Although the SIMNET protocols will tolerate occasional failures by the network to deliver datagrams, these should be allowed to occur only rarely.
- The network should maintain datagram integrity. Any transmission errors that result in the corruption of a datagram should be detected by the network. Corrupted datagrams should not be delivered.
- The network must provide a certain level of performance, which is characterized in terms of throughput and delay. These performance criteria are discussed in the following sections.

It is possible that a computer may receive from the network a datagram that has become corrupted in transit. The network itself provides a means for detecting and reporting most

instances of datagram corruption. One may wish to log these erroneous datagrams as a measure of the network's reliability, possibly providing early warning of any problems with the network. Other than that, the computers participating in a SIMNET exercise can safely ignore erroneous datagrams, discarding them without action. The protocols are sufficiently robust that they can, in most cases, tolerate occasional network errors without human crews participating in the simulation becoming aware of the errors.

4.2 Network throughput

The throughput of the network (the rate at which it can carry information) must be sufficient to support the distributed simulation. However, adequate throughput is not easily predicted, for it depends on many things.

Since Vehicle Appearance PDUs constitute almost all of the data carried by the network during an exercise, the traffic load depends largely on the conditions that result in the transmission of these PDUs. One factor is the manner in which simulated vehicles are operated by their crews. A vehicle in motion usually transmits PDUs faster than one at rest; the exact rate depends on statistics of the vehicle's velocity and acceleration. Another factor is the nature of the model used as a basis for dead reckoning. When a higher order dead reckoning model is used, corrections to a vehicle's dead reckoned state, in the form of Vehicle Appearance PDUs, are usually required less frequently. A third factor is the setting of the thresholds which determine how far a vehicle's actual state is allowed to diverge from its dead reckoned state before a Vehicle Appearance PDU is issued. As these thresholds are relaxed, PDUs will be transmitted less frequently, but the movement of a vehicle will be seen by the crews of other vehicles with less fidelity.

The use of a higher order dead reckoning model may reduce the network traffic load, but each simulator must then perform more computation to dead reckon the state of its own vehicle and the state of every other simulator's vehicle. Any decrease in network traffic load achieved by using higher order derivative information must be weighed against the computational cost of a more complex dead reckoning algorithm, which must be performed by each simulator in proportion to the number of vehicles it is tracking.

The traffic load on the network depends not only on the frequency with which datagrams are transmitted, but also on the sizes of those datagrams. One trade–off concerns the provision of redundant information in the Vehicle Appearance PDUs to save other vehicle simulators from having to recompute it. For example, the vehicle's orientation

about each of three axes is encoded in a Vehicle Appearance PDU as a nine–element rotation matrix, rather than as simply three angles. The resulting increase in network traffic load is more than offset by the savings resulting from simulators receiving the PDU no longer having to compute the rotation matrix themselves to support the common operation of displaying vehicles.

The network resources that provide the throughput necessary for a distributed simulation should not be subject to competition from other demands on the network. In general, this will mean guaranteeing the availability of a certain level of network capacity for a distributed simulation even though that capacity may not be needed at all times.

We have measured the network traffic produced by SIMNET M1 tank simulators and by close–support aircraft simulators, both individually and collectively, when operated in realistic company– and battalion–scale exercises. Although we have measured network traffic under various assumptions of dead reckoning algorithms and discrepancy thresholds, the results described here were obtained using the parameters documented in chapter 3. Each of the simulators we measured will produce up to fifteen Vehicle Appearance PDUs per second if operated in a certain manner (in the case of the M1 simulator, this would require slewing the turret at a high rate). This seldom occurs, however. We have found that an M1 simulator will generate an average of two to three Vehicle Appearance PDUs per second while being actively operated, as during a movement to contact or an attack. Ninety percent of the time, these simulators produce Vehicle Appearance PDUs at a rate of less than four per second. For an active close–support aircraft, about six Vehicle Appearance PDUs per second are produced.

However, when a large population of vehicles are participating together in an exercise, the overall network traffic in terms of Vehicle Appearance PDUs per vehicle is somewhat lower since not all vehicles are being operated actively at any one time. We have found that the network traffic averaged over several companies of tanks, all participating in an exercise at the same time, is about one Vehicle Appearance PDU per second per tank.

4.3 Network delay

By network delay, we mean the amount of time required for a datagram to be carried by the network from one simulator to another. Two characteristics of this delay are important to consider: the average magnitude of the delay, and the degree to which the delay may vary from one datagram to another.

The magnitude of network delay affects the quality of a distributed simulation only in certain ways. Consider a case where a target vehicle is being observed from another vehicle, and the network delays by a fixed amount each target Vehicle Appearance PDU reaching the observer's simulator. The observer will simply always see the target as it was a fixed time earlier. As long as he is passively observing, this delay will not be apparent. However, if the observer is also affecting the target vehicle—such as by striking it with weapons fire, or by acting in a way that causes it to maneuver in response—then the observer may notice the lag due to the network delay.

How much network delay is acceptable can only be determined by considering the application itself—no absolute rule can be given. For distributed simulations involving only ground vehicles, we have found to be acceptable the delays of a few hundred milliseconds that result from transmitting data via satellite channels. However, unless somehow compensated for, this delay would not be acceptable to aircraft pilots attempting to fly in close formation. Note that within a site, the delays experienced by datagrams traversing a local area network are more typically a few milliseconds at most.

The level of fidelity with which a simulator's vehicle is perceived by others is related to the thresholds used by that simulator in determining when to transmit a Vehicle Appearance PDU. This fidelity is also affected by the variance in network delay. If the delay incurred by PDUs travelling from one simulator to another varies widely from one PDU to another, the receiving simulator will be able to dead reckon the sending simulator's vehicle with less than the accuracy dictated by the thresholds. Although the simulation protocol allows delay variance to be somewhat compensated for (using a mechanism described in section 3.4), it is nevertheless desirable to minimize it.

Another consequence of delay variance may be the delivery of datagrams out of sequence. A datagram sent first may be overtaken by one sent second because it encounters a larger delay. The timestamps included in Vehicle Appearance PDUs make it possible to detect those for a particular vehicle that arrive out of sequence. Other portions of the simulation and data collection protocols should not be affected by delay variances of tens, or even hundreds, of milliseconds.

5 PROTOCOL DATA ELEMENTS

Communication carried out according to the governing rules of a protocol involves the exchange of units of data called *protocol data units* (*PDUs*). Each PDU is composed of individual data elements called *fields*. Among the first fields of every PDU is one that identifies the type of the PDU; the header is usually followed by additional fields whose format and meaning depend on the PDU's type.

In this chapter we define data elements that are commonly used in a variety of different types of PDUs of all three SIMNET protocols. We also define in this chapter the concepts of *timer* and *counter*, which are used in describing time–based and repetitive protocol procedures.

5.1 Basic data elements

This section contains definitions of various basic data elements, including how these data elements are represented as communicated bits. Because these elements form parts of many different PDUs, their definitions are collected here for reference in later chapters. A reference appears thus: (§5.1.1).

Data elements are defined in this report using a notation called *Data Representation Notation* (DRN). It is described fully in appendix A.

5.1.1 Angle

Angles, such as the azimuth of a tank's turret relative to the front of the tank, are represented as 32–bit values:

```
type Angle UnsignedInteger (32)
```

The 32–bit value is interpreted as a binary fixed–point fraction with an implied "binary point" to the left of the most significant bit. The fraction expresses the angle as a portion of a full circle, in the range

$$\left[0, 1 - \frac{1}{2^{32}}\right]$$

This method of measuring angles is referred to as Binary Angular Measure (BAM).

5.1.2 Battle Scheme

The battle scheme identifies how force IDs (§5.1.7) and guises (§5.1.23) are being applied in an exercise.

```
type BattleScheme enum (8) {
    battleSchemeOther (0),     -- none of those listed below
    battleSchemeAbsolute,
    battleSchemeRelative
}
```

In an exercise conducted using `battleSchemeAbsolute`, the `distinguished` and `other` object types in the `guises` field of Vehicle Appearance PDUs are assigned as follows:

Vehicle Guises

Force ID	Distinguished	Other
`distinguishedForceID`	US	US
`otherForceID`	Soviet	Soviet
`observerForceID`	US	Soviet
`targetForceID`	Soviet	US

If the battle scheme is battleSchemeRelative, the corresponding assignment is:

Vehicle Guises

Force ID	Distinguished	Other
`distinguishedForceID`	US	Soviet
`otherForceID`	Soviet	US
`observerForceID`	US	US
`targetForceID`	Soviet	Soviet

In the tables above, Force ID refers to the `force` field of the Vehicle Appearance Packet containing the guise. "US" refers refers to an object type representing equipment of US manufacture and "Soviet" refers to an object type representing equipment of Soviet

manufacture. "US" and "Soviet" are for example only. Object types representing the equipment used by any two other opposing forces may be used instead.

5.1.3 Boolean

A Boolean data element is a single bit:

```
type Boolean enum (1) {
    false,
    true
}
```

The value 0 is interpreted as "false", and 1, as "true".

5.1.4 Burst Descriptor

A Burst Descriptor data element describes either a single round of ammunition, or several rounds that are being fired as a burst from a machine gun:

```
type BurstDescriptor sequence {
    projectile          ObjectType,
    detonator           ObjectType,
    quantity            UnsignedInteger (16),
    rate                UnsignedInteger (16)
}
```

The `projectile` field identifies the type of projectile fired, and the `detonator` field identifies the type of detonator (fuze) used. Both are instances of the Object Type data element (§5.1.10), which identifies a type of physical object. The `quantity` field is the number of rounds fired in the burst (1 if only a single shot is described). If `quantity` is greater than 1, the `rate` field is the rate of fire in rounds per second. Otherwise, when `quantity` is 1, the `rate` field contains the following value:

```
constant burstRateIrrelevant 0
```

5.1.5 Event Identifier

An event identifier is a 16–bit serial number generated by a vehicle's simulator. It is associated with a particular event that that vehicle is involved in, such as the firing of a shell or a collision with another vehicle. Each event identifier is unique among all event identifiers generated for that vehicle in the current exercise.

```
type EventID UnsignedInteger (16)
```

Some PDUs contain an event identifier field that is not used in certain cases. If unused, the field should contain the following value:

```
constant eventIDIrrelevant 0
```

5.1.6 Exercise Identifier

An exercise identifier is an 8–bit number that distinguishes one exercise from others occurring on the same network at the same time.

```
type ExerciseID UnsignedInteger (8)
```

Some PDUs contain an exercise identifier field that is not used in certain cases. If unused, the field should contain the following value:

```
constant exerciseIDIrrelevant 0
```

5.1.7 Force Identifier

The participants in an exercise are divided into (usually two) collections of people and equipment, called *forces*. Each force has a unique, 8–bit identifier:

```
type ForceID UnsignedInteger (8)
```

Some PDUs contain a force identifier field that is not used in certain cases. If unused, the field should contain the following value:

```
constant forceIDIrrelevant 0
```

One force identifier has special meaning: vehicles that are assigned to force 1 may view vehicles differently than those assigned to other forces (§2.3). To distinguish this force identifier, it is given a particular name:

```
constant distinguishedForceID 1
```

Three other forces have been defined:

```
constant otherForceID          2
constant observerForceID       3
constant targetForceID         4
```

By convention, these force IDs have the following meanings:

otherForceID	the opponent of distinguishedForceID
observerForceID	appears to be friendly to both combatants
targetForceID	appears to be an enemy to both combatants

The manner in which vehicle guises (§5.1.23) are assigned to these force IDs is governed by the battle scheme (§5.1.2).

5.1.8 Munition Quantity

An amount of some munition is represented by a Munition Quantity data element, which defines the type of munition and the quantity of it:

```
type MunitionQuantity sequence {
    munition            ObjectType,
    quantity            Float (32)
}
```

The `munition` field of this data element identifies the type of munition (§5.1.10), and the `quantity` field specifies its quantity. The units in which the quantity is measured vary according to the type of munition described. In general, ammunition is measured in rounds and fuel is measured in gallons.

5.1.9 Object Identifie

Object identifiers serve to uniquely identify individual objects generated by simulators using a state update procedure(§7.3.3). Object identifiers are used in place of vehicle identifiers(§5.1.24) for objects that are not vehicles, for example: mine fields. An object identifier is composed of two parts: a simulation address, which identifies the simulator modeling that object(§5.1.13) and an object number, which uniquely identifies the object within that simulator and exercise.

```
type ObjectID  sequence {
    simulator           SimulationAddress,      -- of object's simulator
    object              UnsignedInteger (16)    -- unique for that simulator
}
```

The format of the simulator field is defined in §5.1.13.

No two objects present in the same exercise may have identical object identifiers. No object and vehicle present in the same exercise may share both the same simulation address and the same object or vehicle number (see §5.1.24).

5.1.10 Object Type

Physical objects present in the simulated world include vehicles and munitions. Each object has a particular type—for example, it is an M1 tank, or a Hellfire missile. An object's type is described by a 32–bit code:

```
type ObjectType UnsignedInteger (32)
```

Object type values are defined in appendices B and C.

Some PDUs contain an object type field that is not used in certain cases. If unused, the field should contain the following value:

```
constant objectIrrelevant 0
```

5.1.11 Organizational Unit

Each simulated vehicle may be associated with a series of organizational units, such as a company, a battalion, etc. An Organizational Unit data element is used to refer to a particular unit at any level in the organizational hierarchy, and to identify the larger units of which it is also a member.

```
type OrganizationalUnit sequence {
    force              ForceID,
    organizationType   OrganizationType,
    hierarchy          array (organizationalLevels) of UnitIdentifier
}

constant organizationalLevels 9
```

The grossest subdivision of units is into collections called *forces* (§5.1.7); the `force` field identifies that to which the unit belongs. The `organizationType` field describes how that force is organized, and determines how the `hierarchy` field is to be interpreted. Currently, two values are defined for the `organizationType` field:

```
type OrganizationType enum (8) {
    organizationIrrelevant,
    organizationArmy
}
```

The `organizationType` value `organizationIrrelevant` (defined as 0) is used when no organizational hierarchy is to be specified. In such cases, all elements of the `hierarchy` array should contain zeros.

In other cases, the elements of the `hierarchy` array identify a succession of nested units, beginning from the top of an organizational hierarchy. Each array element may specify a single unit, by number and type:

```
type UnitIdentifier sequence {
    unitNumber          UnsignedInteger (8),
    unitType            UnitType
}

constant unitNumberIrrelevant 0

type UnitType enum (8) {
    unitTypeIrrelevant (0),
    unitTypeArmy,
    unitTypeBattalion,
    unitTypeBattery,
    unitTypeBrigade,
    unitTypeCompany,
    unitTypeCorps,
    unitTypeDivision,
    unitTypeFlight,
    unitTypeGroup,
    unitTypePlatoon,
    unitTypeRegiment,
    unitTypeSection,
    unitTypeSquad,
    unitTypeSquadron,
    unitTypeTaskForce,
    unitTypeTeam,
    unitTypeTroop,
    unitTypeWing
}
```

Not all elements of the array are used in each case; those that are not used to specify a unit contain zeros.

The `organizationType` value `organizationArmy` is specified when the `hierarchy` array identifies a unit within an organizational structure that is like that of the U.S. Army or the Soviet Army. In this case, the elements of the `hierarchy` array are used in the following way tc describe the unit:

0	the army
1	the corps within that army
2	the division within that corps
3	the brigade, regiment, or group within that division
4	the battalion or squadron within that brigade
5	the company, team, battery, or troop within that battalion or squadron
6	the platoon within that company, etc.
7	the section within that platoon
8	the squad within that section

Within this framework, array elements that are not relevant contain zeros. If an entire company is being referred to, for example, then the last three elements of the array (corresponding to platoon, section, and squad) contain zeros.

5.1.12 Repair Type

The simulation protocol includes a mechanism for carrying out simulated repairs on disabled vehicles. Although the types of repairs that may be performed depend on the kind of vehicle being repaired, all repairs are identified by single, 16–bit integers:

```
type RepairType UnsignedInteger (16)
```

A particular repair code, 0, has the effect of correcting all of the repaired vehicle's disabilities:

```
constant repairEverything 0
```

Other repair codes are specific to the type of vehicle being repaired, and the type of simulator modeling that vehicle. Appendix D defines repair codes for those vehicle simulators developed under the SIMNET program.

5.1.13 Simulation Address

Each simulator that participates in the distributed simulation internet has a globally unique address. This address is composed of two parts: a site identifier, which specifies the site where the entity resides; and a host number, that distinguishes the simulator's host computer from among other hosts at the same site.

```
type SimulationAddress sequence {
    site                SiteID,
    host                UnsignedInteger (16)
}
```

The Site ID data element is defined in §5.1.15. If the `host` field is not relevant in a particular case, it should contain 0:

```
constant hostIrrelevant 0
```

5.1.14 Simulator Type

Each type of simulation system participating in the distributed simulation internet is described by a simulator type code. The value 0 is used in cases where no other simulator type code is applicable.

```
type SimulatorType enum (16) {
    simulatorUnknown,
    simulator_SIMNET_MCC,              -- SIMNET MCC system
    simulator_SIMNET_SAF,              -- SIMNET SAF system
    simulator_SIMNET_M1,               -- SIMNET M1 (Abrams) simulator
    simulator_SIMNET_M2,               -- SIMNET M2/3 (Bradley) simulator
    simulator_SIMNET_FRED,             -- SIMNET rotary-wing simulator
    simulator_SIMNET_FWA,              -- SIMNET fixed-wing simulator
    simulator_SIMNET_FAAD_LOS_H,       -- SIMNET FAAD-LOS-H simulator
    simulator_SIMNET_STEALTH,          -- SIMNET Stealth simulator
    simulator_SIMNET_DI,               -- SIMNET Dismounted Infantry
    simulator_AGPT_LEO2,               -- AGPT Leo2 simulator
    simulator_AGPT_ULF,                -- AGPT Stealth simulator
    simulator_AGPT_DATA_LOGGER,        -- AGPT Data Logger
    simulator_SIMNET_LOSAT             -- SIMNET Losat vehicle simualator
}
```

5.1.15 Site Identifier

Each site of the distributed simulation internet is assigned a unique, 16–bit identifier:

```
type SiteID UnsignedInteger (16)
```

The value 0 is used to indicate that this identifier is irrelevant in a particular case:

```
constant siteIrrelevant 0
```

5.1.16 Target Descriptor

A simulator issues a report when its vehicle fires upon another vehicle, or uses its laser to range to another vehicle. In reporting the event, the simulator includes a description of what it knows of the vehicle being fired upon or ranged to. This description is a Target Descriptor data element:

```
type TargetDescriptor sequence {
    targetType          TargetType,
                        unused (8),
    vehicleID           VehicleID
}
```

The `targetType` field specifies how much the simulator knows about the target:

```
type TargetType enum (8) {
    targetUnknown,                  -- the target vehicle is not known
    targetIsNotVehicle,             -- target known, but not a vehicle
    targetIsVehicle                 -- target known, and a vehicle
}
```

If the target is known and it is a vehicle, that vehicle's identifier is included in the `vehicleID` field (§5.1.24). Otherwise, that field contains zeros.

5.1.17 Terrain Database Identifier

When a simulator is activated by another computer system, the terrain database it is to use for its simulation is described by a Terrain Database ID data element:

```
type TerrainDatabaseID sequence {
    terrainName         array (maxTerrainNameLength) of Character (8),
    terrainVersion;     UnsignedInteger (16)
}

constant maxTerrainNameLength 14
```

The terrainName field contains a mixed–case alphabetic string, encoded in ASCII and padded with null (0) characters to a length of 14 octets. "Knox" and "Graf" are examples of terrain database names. The terrainVersion field contains a positive version number, or 0 if the most recent version of the database is to be used:

```
constant latestTerrainVersion 0
```

5.1.18 Time

A Time data element represents a date and time as a count of the seconds elapsed since 0 GMT, 1 January 1970:

```
type Time UnsignedInteger (32)
```

5.1.19 Vehicle Capabilities

A simulated vehicle is advertised on the network as being capable of supplying certain services to other simulated vehicles. These are described by a Vehicle Capabilities data element:

```
type VehicleCapabilities sequence {
    ammunitionSupply    Boolean,
    fuelSupply          Boolean,
    recovery            Boolean,
    repair              Boolean,
                        unused (28)
}
```

The ammunitionSupply field and fuelSupply field are true if the vehicle is capable of supplying ammunition and fuel, respectively. The recovery field is true if the vehicle is capable of recovering (towing) other vehicles. The repair field is true if the vehicle is capable of carrying out repairs to other vehicles.

5.1.20 Vehicle Class

Each vehicle is classified according to how many independently movable parts it has (such as turrets and gun barrels), and according to what algorithm should be used to dead reckon its appearance. The classifications are called *vehicle classes*. They are identified by 8–bit integers:

```
type VehicleClass enum (8) {
    vehicleClassIrrelevant,          -- class irrelevant
    vehicleClassStatic,              -- static class
    vehicleClassSimple,              -- simple class
    vehicleClassTank                 -- tank class
}
```

The value `vehicleClassIrrelevant` is used in cases where the value of a Vehicle Class data element is not relevant.

5.1.21 Vehicle Component

When a vehicle is struck by weapons fire, the damage suffered may depend on what part of the vehicle was struck. A specification of a vehicle component is one form in which the location of a hit is communicated among simulators. A vehicle component is identified by the following data element:

```
type VehicleComponent enum (16) {
    vehicleComponentIrrelevant,      -- none of those listed below
    hullComponent,
    turretComponent
}
```

The value `turretComponent` is applicable to turreted vehicles, such as tanks. The value `hullComponent` is applicable to all vehicle types. Values for additional components may be defined as needed.

5.1.22 Vehicle Coordinates

A location may be specified, with reference to a particular vehicle's coordinate system, by a Vehicle Coordinates data element. Each coordinate is a floating–point number, measuring a distance in meters along one axis of the vehicle's coordinate system.

```
type VehicleCoordinates array (3) of Float (32)
```

The three elements of the array represent, in order, the X coordinate, the Y coordinate, and the Z coordinate.

5.1.23 Vehicle Guises

The basic appearance of a vehicle is described by an object type code (§5.1.10) that identifies a particular type of vehicle, such as an M1 or a T72. Some applications of distributed simulation require that each vehicle have two alternate appearances: one when viewed by some observers, and a different one when viewed by others. Other applications require that each vehicle appear identical to all observers.

To support both kinds of applications, two object type codes are used to report a vehicle's appearance. They are packaged as a Vehicle Guises data element:

```
type VehicleGuises sequence {
    distinguished       ObjectType,
    other               ObjectType
}
```

Which of the object type codes determines the basic appearance of the vehicle depends on what force the observer is assigned to. If the vehicle is being observed by someone assigned to force 1 (represented by the `distinguishedForceID` constant), then the vehicle's appearance is that specified by the `distinguished` field. Otherwise, it is that specified by the `other` field.

5.1.24 Vehicle Identifier

Every vehicle participating in an exercise has associated with it a unique vehicle identifier. A vehicle identifier is composed of two parts: a simulation address, which identifies the simulator modeling that vehicle (§5.1.13); and a vehicle number, which distinguishes the vehicle from others generated by the same simulator in the same exercise.

```
type VehicleID sequence {
    simulator           SimulationAddress,
    vehicle             UnsignedInteger (16)
}
```

The format of the simulator field is defined in §5.1.13

The two parts of the vehicle identifier, taken together, form a globally unique value: the `simulator` component uniquely identifies a simulator, and the `vehicle` component uniquely identifies a vehicle modeled by that simulator. Vehicle identifiers share the

same number space as object identifiers (§5.1.9). Thus, no object and vehicle present in the same exercise may share both the same simulation address and the same object or vehicle number.

Some PDUs contain a vehicle identifier field that is not used in certain cases. An example is the vehicle identifier field of a Target Descriptor ID, which is unused if the target is not known or is not a vehicle. In such cases, all components of the vehicle identifier should contain zeros.

No vehicle is assigned a vehicle identifier whose `vehicle` component is zero.

5.1.25 Vehicle Marking

A vehicle may have a marking that is visible when viewed from other vehicles under certain circumstances. The marking may, for example, be ship's name or a tank's bumper number. A Vehicle Marking data element describes it:

```
type VehicleMarking sequence {
    characterSet        CharacterSetType,
    text                array (maxVehicleMarkingLength) of
                          UnsignedInteger (8)
}

constant maxVehicleMarkingLength 11
```

The `characterSet` field identifies the character set according to which the text of the marking should be interpreted and displayed. Currently, one character set is defined:

```
type CharacterSetType enum (8) {
    asciiCharacterSet
}
```

The `text` field contains a string of from 0 to 11 characters, padded with null (0) characters to a length of 11.

5.1.26 Vehicle Status

The operational status of a vehicle, the health of each of its subsystems, and the quantities of the various munitions it carries are all represented by a single Vehicle Status data element:

```
type VehicleStatus sequence {
    vehicleType         ObjectType,
    odometer            Float (32),
    age                 UnsignedInteger (8),
                        unused (24),
    failures            VehicleSubsystems,
    specific            VehicleSpecificStatus
}
```

The `vehicleType` field specifies the type of vehicle described by the data element (§5.1.10). The `age` field contains the vehicle's age, in years, and the `odometer` field contains its lifetime travel, in meters.

The `failures` field describes the operational status of the vehicle (§5.1.27). It identifies the specific subsystems of the vehicle that are simulated, and indicates those that have failed. A failed subsystem is represented by the value false for the appropriate Boolean, and an operational subsystem, by the value true. The `failures` field also indicates, by means of five Boolean fields, whether the vehicle has suffered a catastrophic kill, or a partial kill classified as involving mobility, firepower, communication, or non–critical subsystems.

The `specific` field represents additional, vehicle–specific information, including the quantities of various kinds of munitions on board. The format of this field depends on the type of vehicle being described, and the manner in which that vehicle is simulated:

```
type VehicleSpecificStatus sequence {
    category            SpecificStatusCategory,
                        unused (16),
    specific            choice (category) of {

      when (genericVehicleStatus)
        generic         GenericVehicleStatus,

      when (simnetM1Status)
        m1              SIMNET_M1_Status,

      when (simnetM2Status)
        m2              SIMNET_M2_Status,

      when (simnetFAADStatus)
        faad            SIMNET_FAAD_LOS_H_Status,

      when (agptLeo2Status)
        leo2            AGPT_Leo2_Status
    }
}
```

The `category` field classifies the vehicle according to the format of its vehicle–specific information. Currently, three categories are defined:

```
type SpecificStatusCategory enum (16) {
    genericVehicleStatus,
    simnetM1Status,
    simnetM2Status,
    simnetFAADStatus,
    agptLeo2Status
}
```

The `genericVehicleStatus` category is provided for describing any vehicle in a generic, but limited, manner. Other categories exist for describing specific types of simulated vehicles in greater detail. There are presently two such categories, which are used for describing vehicles simulated by the SIMNET M1 and the SIMNET M2/3 simulators. The SIMNET_M1_Status and SIMNET_M2_Status data elements used in these cases are defined in appendix D.

If the vehicle–specific status is being described in a generic manner (admittedly, a contradiction of terms), then the following representation is used:

```
type GenericVehicleStatus sequence {
    enginePower         UnsignedInteger (8),
    batteryVoltage      UnsignedInteger (24),
    stores              array (maxGenericVehicleStores) of
                          MunitionQuantity
}

constant maxGenericVehicleStores 6
```

The `enginePower` field specifies the percent of full power the vehicle's engine is able to produce. The voltage of the vehicle's battery (in millivolts) is specified by the `batteryVoltage` field. The `stores` array lists the quantities of various kinds of munitions carried by the vehicle (§5.1.8). If fewer than six different kinds of munitions are carried, then the first elements of the array are used to represent munitions, and the remaining elements are filled with zeros.

5.1.27 Vehicle Subsystems

A Vehicle Subsystems data element specifies a set of vehicle subsystems. It is used, for example, to indicate those subsystems that have failed in a particular vehicle, or those that have changed as a result of a repair.

```
type VehicleSubsystems sequence {
    category            SubsystemsCategory,
    operationalSummary  Boolean,
    mobilitySummary     Boolean,
    firepowerSummary    Boolean,
    communicationSummary Boolean,
    noncriticalSummary  Boolean,
                        unused (11),
    subsystems          choice (category) of {
      when (airVehicleSubsystems)
        air             AirVehicleSubsystems,
      when (groundVehicleSubsystems)
        ground          GroundVehicleSubsystems
    }
}
```

The subsystems of the vehicle are represented in two forms: a summary is provided by the five Boolean fields, and detailed information is provided by the `subsystems` field. The meaning of the five Boolean fields is as follows:

`operationalSummary`	summarizes those subsystems that determine whether the vehicle is at all operational. If, for example, the Vehicle Subsystems data element is being used to indicate which subsystems of a vehicle have failed, this bit is 1 if the vehicle has suffered a catastrophic kill.
`mobilitySummary`	summarizes those subsystems that provide mobility.
`firepowerSummary`	summarizes those subsystems that provide firepower.
`communicationSummary`	summarizes those subsystems that support communication.
`noncriticalSummary`	summarize other, noncritical subsystems.

More detailed information on a vehicle's subsystems is contained in the `subsystems` field, which identifies individual subsystems of the vehicle. The format of this field depends on the category of vehicle being described. Presently, two categories of vehicle are defined to allow the data element to be used to describe the subsystems of air and ground vehicles in a generic manner. Other categories may be defined as necessary.

```
type SubsystemsCategory enum (16) {
    airVehicleSubsystems (1),
    groundVehicleSubsystems (2)
}
```

For a vehicle of either generic category, the Vehicle Subsystems data element includes individual fields corresponding to many specific subsystems. No one vehicle, however, is expected to have all of these subsystems. When the data element is used to describe a particular vehicle, the fields used are only those that correspond to subsystems that the vehicle actually has. For each possible subsystem, the data element indicates both (a) whether the subsystem exists in the simulated vehicle, and (b) if it exists, the status of that subsystem.

The set of generic subsystems is divided into logical groups: electronic subsystems, motive power subsystems, electrical and hydraulic power subsystems, etc. Each group is represented by a collection of up to 32 Boolean data elements, with one Boolean representing each subsystem. A Vehicle Subsystems data element contains two instances of each group, organized as two elements of an array. The first element indicates which subsystems of the group exist; a Boolean in the first element is true if the corresponding

subsystem exists. The second element indicates the status of those that do exist; the interpretation of these Booleans depends on the context in which the Vehicle Subsystems data element is being used. If a Boolean of the first element indicates that a particular subsystem does not exist, then the corresponding Boolean of the second element is always false.

The following notation defines these data elements:

```
type AirVehicleSubsystems sequence {
    electronic          array (2) of ElectronicSubsystems,
    motive              array (2) of MotiveSubsystems,
    power               array (2) of PowerSubsystems,
    weapon              array (2) of WeaponSubsystems,

    -- Specific to air vehicles:
    airframe            array (2) of AirframeSubsystems,
    cockpit             array (2) of CockpitSubsystems
}

type GroundVehicleSubsystems sequence {
    electronic          array (2) of ElectronicSubsystems,
    motive              array (2) of MotiveSubsystems,
    power               array (2) of PowerSubsystems,
    weapon              array (2) of WeaponSubsystems,

    -- Specific to ground vehicles:
    chassis             array (2) of ChassisSubsystems,
    turret              array (2) of TurretSubsystems
}

define subsystemExists 0                    -- first element of array
define subsystemStatus 1                    -- second element of array
```

The constants `subsystemExists` and `subsystemStatus` are index values defined for the two–element arrays.

The following notation defines the collections of Booleans that represent individual subsystems:

```
type AirframeSubsystems sequence {

    -- Airframe components:
    airframeMajor       Boolean,
    leftWing            Boolean,
    rightWing           Boolean,
                        unused (5),

    -- Control surfaces:
    pitchControl        Boolean,
    rollControl         Boolean,
    yawControl          Boolean,
    flaps               Boolean,
    airBrakes           Boolean,
                        unused (3),

    -- Landing gear:
    landingGearMajor    Boolean,
    noseWheel           Boolean,
    leftWheel           Boolean,
    rightWheel          Boolean,
                        unused (12)
}

type ChassisSubsystems sequence {

    -- Final drive subsystems:
    finalDriveMajor     Boolean,
    leftTrack           Boolean,
    rightTrack          Boolean,
    leftFrontWheel      Boolean,
    rightFrontWheel     Boolean,
    leftRearWheel       Boolean,
    rightRearWheel      Boolean,
    serviceBrake        Boolean,
    parkingBrake        Boolean,
                        unused (7),

    -- Hull subsystems:
    hullMajor           Boolean,
                        unused (7),

    -- Driver subsystems:
    driversVisionBlocks Boolean,
                        unused (7)
}
```

```
type CockpitSubsystems sequence {
    cockpitMajor        Boolean,
                        unused (31)
}

type ElectronicSubsystems sequence {

    -- Communication subsystems:
    communicationMajor  Boolean,
    radioAntenna        Boolean,
    intercom            Boolean,
    radio               Boolean,
                        unused (12),
    -- Sensor subsystems:
    laserRangefinder    Boolean,
    electroOpticalMajor Boolean,
    eoFLIR              Boolean,
    eoDATV              Boolean,
    eoCtlHandle         Boolean,
    radarMajor          Boolean,
    radarTransceiver    Boolean,
    radarTracking       Boolean,
    radarNetwork        Boolean,
    navigationMajor     Boolean,
                        unused (6)
}

type PowerSubsystems sequence {

    -- Electrical power subsystems:
    priElectrical       Boolean,
    priDistributionBox  Boolean,
    secElectrical       Boolean,
    secDistributionBox  Boolean,
    alternator          Boolean,
    generator           Boolean,
                        unused (10),

    -- Hydraulic power subsystems:
    priHydraulic        Boolean,
                        unused (15)
}
```

```
type MotiveSubsystems sequence {

    -- Engine subsystems:
    engineMajor         Boolean,
    pilotRelay          Boolean,
    starter             Boolean,
    oilFilter           Boolean,
    oilLeak             Boolean,
    airCleaner          Boolean,
    coolantLeak         Boolean,
    fuelFilter          Boolean,
    fuelXferPump        Boolean,
                        unused (7),

    -- Drive train subsystems:
    transmissionMajor   Boolean,
    transFluidFilter    Boolean,
    transFluidLeak      Boolean,
    leftGearbox         Boolean,
    rightGearbox        Boolean,
    universalJoint      Boolean,
                        unused (10)
}

type TurretSubsystems sequence {
    turretMajor         Boolean,
    turretTraverse      Boolean,
    stabilization       Boolean,
    gunnersCtlHandle    Boolean,
    cmdrsCtlHandle      Boolean,
    turretPositionInd   Boolean,
    turretSlopeInd      Boolean,
    gunnersPrimarySight Boolean,
    gunnersSecondarySight Boolean,
    gpsExtension        Boolean,
    gunnersVisionBlocks Boolean,
    cmdrsVisionBlocks   Boolean,
    loadersPeriscope    Boolean,
    cmdrsPeriscope      Boolean,
                        unused (18)
}
```

```
type WeaponSubsystems sequence {
    priGunMajor         Boolean,
    priGunMount         Boolean,
    priGunElevation     Boolean,
    priGunTraverse      Boolean,
    priGunMisfire       Boolean,
    secGunMajor         Boolean,
    secGunMount         Boolean,
    secGunElevation     Boolean,
    secGunTraverse      Boolean,
    secGunMisfire       Boolean,
    missileMajor        Boolean,
    launcher1           Boolean,
    launcher2           Boolean,
                        unused (19)
)
```

5.1.28 Velocity Vector

The velocity of a vehicle or projectile is represented by a Velocity Vector data element. It specifies the component of the velocity that is parallel to each of the world coordinate system's three axes, in meters per second:

```
type VelocityVector array (3) of Float (32)
```

The three elements of the array represent, in order, the velocity components parallel to the X axis, the Y axis, and the Z axis.

5.1.29 World Coordinates

A location in the simulated world is defined by a set of three coordinates. Each coordinate is a floating–point number, measuring a distance in meters along one axis of the world coordinate system (§2.4):

```
type WorldCoordinates array (3) of Float (64)
```

The three elements of the array represent, in order, the X coordinate, the Y coordinate, and the Z coordinate.

5.1.30 XY Coordinates

XY Coordinates represent a location in the X-Y plane of the world coordinate system (§2.4). Each coordinate is represented as a floating-point number, measuring distance along a coordinate axis in units of meters.

```
type XYCoordinates array (2) of Float(64)
```

The first element of the array represents the X coordinate and the second element of the array represents the Y coordinate.

5.2 Timers and counters

Some protocol procedures call for the repeated, periodic transmission of PDUs. For example, the data collection protocol requires that each crewed vehicle simulator transmit a Vehicle Status PDU every 30 seconds. We make use of a conceptual device called a *timer* to describe the periodicity of a repeated transmission. A second conceptual device, called a *counter*, is used to describe how many times the transmission is repeated. The use of counters and timers is explained in the following paragraphs.

A *timer* is a variable used to determine the instants at which a periodically transmitted PDU must be sent. When the PDU is first sent, the timer is set to the number of seconds that must elapse before the PDU is sent again. The timer is then decremented by one unit each second. When the timer reaches zero the PDU is resent, the timer is set back to its original value, and the cycle repeats.

Some protocol procedures call for repeating the transmission of a PDU up to a specified maximum number of times. This is usually done when the sender is not sure that a PDU is being correctly conveyed to its recipient, and so must resend it until confirmation is received. A *counter* is a variable used to keep track of the number of times the PDU has been sent. When the PDU is first sent, the counter is set to the maximum number of times that the PDU may be sent. When the PDU is resent (usually after a timer has expired) the counter is decremented by one unit, and, if it hasn't reached zero, the PDU is sent again.

Several timers and counters are involved in the definition of the SIMNET protocols. The durations of the timers and the initial, maximum values of the counters are given names like `transactionRetryTime` and `transactionRetryCount`. The optimal values

of these depend on factors such as the frequency with which information should be made available for later analysis or recovery of an exercise, and the reliability and delay characteristics of the distributed simulation internet. The values chosen for the present applications and implementations of the SIMNET protocols are summarized in appendix F. Where a timer or counter is mentioned in the following chapters, we specify its typical value for readability but also include its name in parentheses for reference to appendix F.

6 ASSOCIATION PROTOCOL

In chapter 2 the association protocol was introduced as a protocol providing communication services used to support both the simulation protocol and the data collection protocol. This chapter defines the association protocol in terms of the services it provides, and the protocol procedures used to implement those services.

6.1 Architecture

The ISO Basic Reference Model for Open Systems Interconnection [8] is the architectural framework within which the SIMNET protocols are defined. The communication service underlying the SIMNET protocols provides functions associated with the physical, data link, and network layers of the OSI model. The requirements for these services are discussed in chapter 4.

The association protocol is designed to offer a streamlined composite of the specific transport, session, and application layer services that are required by both the simulation and data collection protocols.[4] Although pieces of the transport, session, and application layers are included in this composite, it resides in the application layer as a sublayer. There it is viewed, in the terms of the OSI model, as implementing common application service elements—i.e., services that are shared among multiple application protocols.

By combining several functions into a single protocol, the association protocol implements those functions in the most efficient manner possible. Although the association protocol can be carried by underlying presentation, session, and transport protocols, those underlying protocols are not required. In its operation the association protocol requires no services particular to the presentation, session, or transport layers, other than connectionless–mode transmission of data to multiple recipients. Thus, the association protocol may be implemented directly from services of the network or data link layers to obtain maximum efficiency.

The sublayer containing the association protocol is called the *association sublayer*, and the service it provides is the *association service*. Distributed simulation is implemented

---

4 No services specific to the presentation layer are required by the SIMNET protocols. The presentation of application data is defined by the Data Representation Notation documented in appendix A.

as a pair of protocols—the simulation protocol and the data collection protocol—that make use of the association service for their communication requirements, and reside in a sublayer above it called the *simulation sublayer*. This layered structure is represented in figure 6–1.

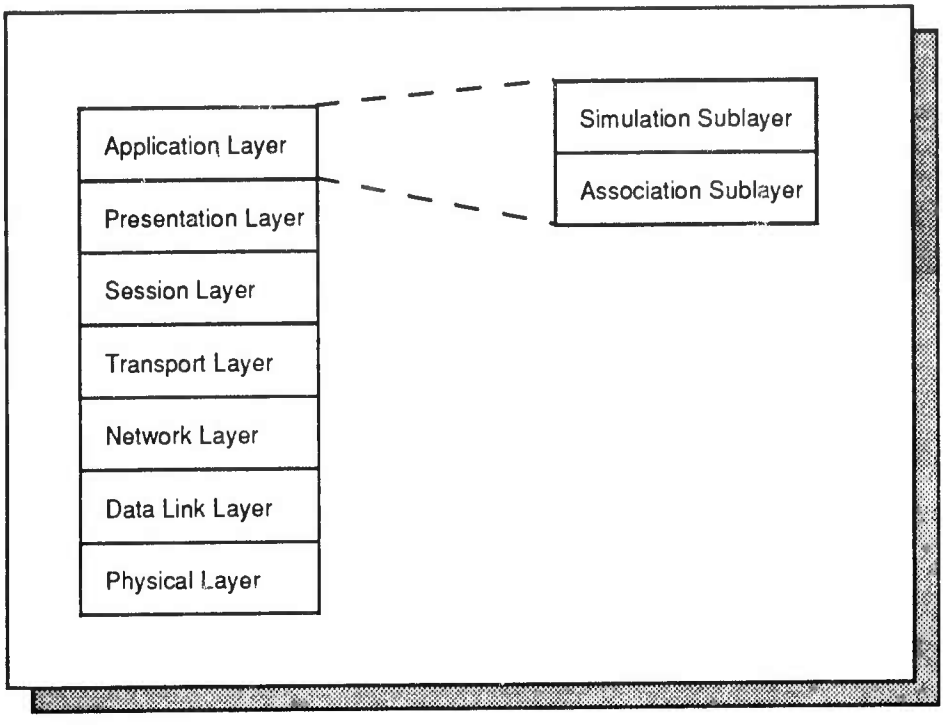


Figure 6–1. The application layer of the OSI reference model is divided into two sublayers: an association sublayer providing common application service elements, and a simulation sublayer implementing distributed simulation.

The association sublayer spans all simulators at all sites to provide a communication service among them, as shown in figure 6–2. The sublayer includes a component within each simulator, called the *association entity*. The simulation sublayer, in turn, spans all simulators at all sites to provide a distributed simulation. The component of the simulation sublayer within each simulator is called the *simulation entity*. A simulation entity obtains services from its local association entity via a single association service access point. The address of that service access point we call a *simulation address*.

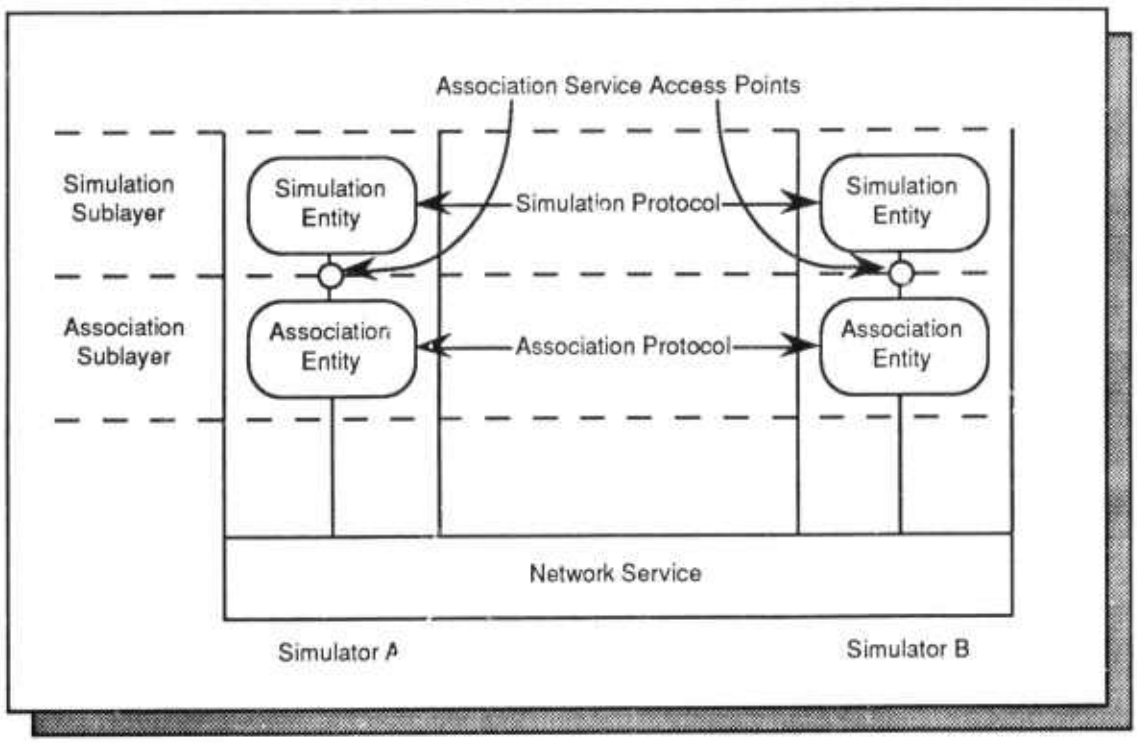


Figure 6–2. The components of the simulation sublayer and the association sublayer within each simulator are called *simulation entities* and *association entities*. Association entities communicate according to *association protocol*, using the network service. Simulation entities communicate according to *simulation protocol*, using the association service. The interface between a simulation entity and its local association entity is an *association service access point*; it has an address called a *simulation address*.

6.2 Service elements

The users of the association sublayer are simulation entities, residing in the simulation sublayer and communicating by means of the association service. These are the key elements of the service provided to simulation entities:

Datagrams. The association service will convey a unit of data from an originator to one or more recipients. This service element is called the *datagram service.*

Transactions. The association service will mediate an interaction that involves the transfer of a unit of data from an originator to a recipient, and the return of an associated unit of data from the recipient to the originator. The service must also accommodate the

unacknowledged receipt of both data units by any number of other simulation entities. This service element is called the *transaction service*.[5]

Detecting and overcoming network failures. The association service must provide a minimum probability of successful data transfer for the transaction service (whereas the reliability of the underlying networks is assumed to be sufficiently high for the uses of the datagram service). If an underlying subnetwork has an insufficient probability of successful delivery, then the association sublayer compensates by retrying certain data transfers.

User protocol. Along with any unit of data it transfers, the association service will convey a user protocol number provided by the sending simulation entity. This protocol number is conveyed so that simulation entities can indicate to each other the simulation sublayer protocol to which a unit of data pertains. The protocol numbers are 8–bit integers, with values that must be agreed upon by corresponding simulation entities. This is done by assigned a unique global identifier to each protocol that uses the association service.

Addressing. The association service provides simulation entities with a consistent method of addressing each other, independent of the addressing schemes of the various subnetworks employed. A single simulation entity is addressed at each simulator by an address that consists of two components:

- A 16–bit integer that uniquely identifies the site at which the simulator resides. A unique global site identifier must be assigned to each site in the distributed simulation internet.
- A 16–bit integer that uniquely identifies the particular simulator within that site.

---

5 The transaction service provided by the association sublayer is a hybrid of transport and application layer functionality. Reliable end–to–end transfer, involving the possible retransmission of data, is a transport layer function. Providing a request/response transaction mechanism is an application layer function. The association PDU conveying a response is used to efficiently provide both a transport layer acknowledgement and an application layer transaction response. Otherwise, if separate transport layer acknowledgements were utilized, a total of four transport PDUs would be required (or more in the case of retransmissions).

Multicast groups. The association service allows simulation entities to subscribe to multicast groups. A unit of data directed to a multicast group will be received by all of the simulation entities currently subscribing to that group.[6]

Each multicast group is identified by an 8–bit integer. By convention, the value 0 is used to designate a multicast group that includes all simulation entities. Each simulation entity indicates to the association service those additional multicast groups to which it wishes to subscribe.

Blocking. To make the most efficient use of the network service, the association service may place multiple association protocol data units in a single network datagram.

6.3 Service required from lower layers

The association protocol uses the services of lower layers in order to provide enhanced services for the simulation sublayer. The services used are those described in chapter 4. The association protocol may be implemented on top of any protocols at any layer that can support the service requirements of chapter 4. Appendix E defines how the association protocol is implemented on IEEE 802 networks.

The lower layer service must provide for the transfer of at least 256 octets of user data in a single datagram. For efficiency, a single datagram may contain multiple association protocol data units, provided all association PDUs are being addressed to the same recipients. The recipients of a datagram must be able to unbundle the PDUs contained therein. The association protocol allows this by including in each PDU a field from which the length of that PDU can be determined. This field is used by recipients to determine where successive PDUs lie within the datagram.

Some networks, including Ethernet, require that datagrams be of some minimum length. To accommodate this requirement, the association protocol allows a datagram to be filled out to a minimum length by including in it a PDU whose sole purpose is to occupy space in the datagram. This padding PDU, which appears as the last PDU in a datagram, is inserted only when necessary.

---

6 The association of users into one or more multicast groups is a special session layer function provided by the association sublayer to eliminate the need for a full session service.

6.4 Service provided by the association sublayer

The service provided by the association layer is defined here in terms of primitives that characterize the interaction between the association layer and its users. This method follows the descriptive conventions documented in ISO TR 8509 [9]. For a given primitive, the presence of each parameter is described by one of the following symbols:

M The parameter is mandatory.

U The use of the parameter is a service–user option.

(=) The value of the parameter is identical to the corresponding parameter in the interaction described by the preceding related service primitive.

6.4.1 Group subscription service

A simulation entity specifies to the association service those multicast groups of which it is to be a member. This is a local interaction between the simulation entity and its related association entity.

There are two primitives associated with this service:

A–Subscribe.req

A–Subscribe.conf

The sequence in which these primitives are used is illustrated in figure 6–3.

The simulation entity issues an A–Subscribe request to the association sublayer to indicate that it is subscribing to a multicast group, or unsubscribing from a group previously subscribed to. The association sublayer responds with an A–Subscribe confirmation.

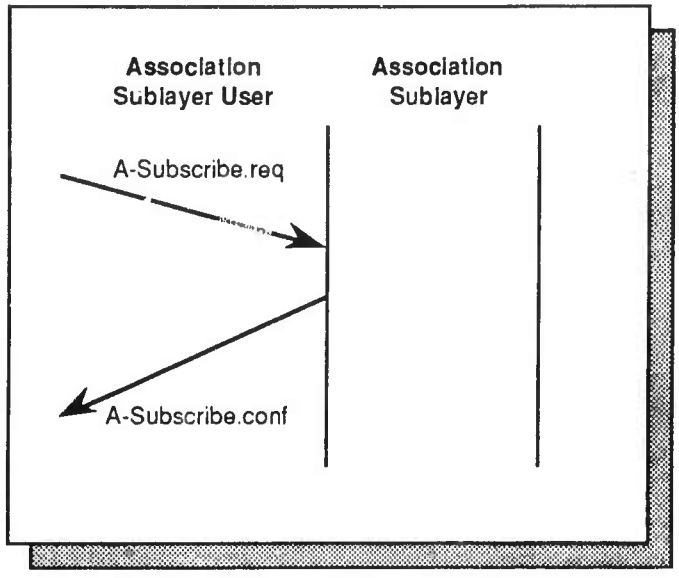


Figure 6–3. The sequence of service primitives associated with the group subscription service.

The parameters associated with these primitives are:

Parameter Name	Req	Conf	Description
multicast–group	M		the multicast group for which the user wishes to change its subscription
subscribe	M		whether the user wishes to subscribe to, or unsubscribe from, the multicast group
result		M	indicates success or failure of the association sublayer operation

All simulation entities are members of multicast group 0, and may not withdraw from it. The composition of other multicast groups (1 through 255), however, is determined through the used of the A–Subscribe service element by individual simulation entities.

6.4.2 Datagram service

The datagram service is used to transfer a unit of data from an originator to a multicast group of one or more recipients. This is a "non–confirmed" service.

There are two primitives associated with this service:

A–Datagram.req

A–Datagram.ind

The sequence in which these primitives are used is illustrated below:

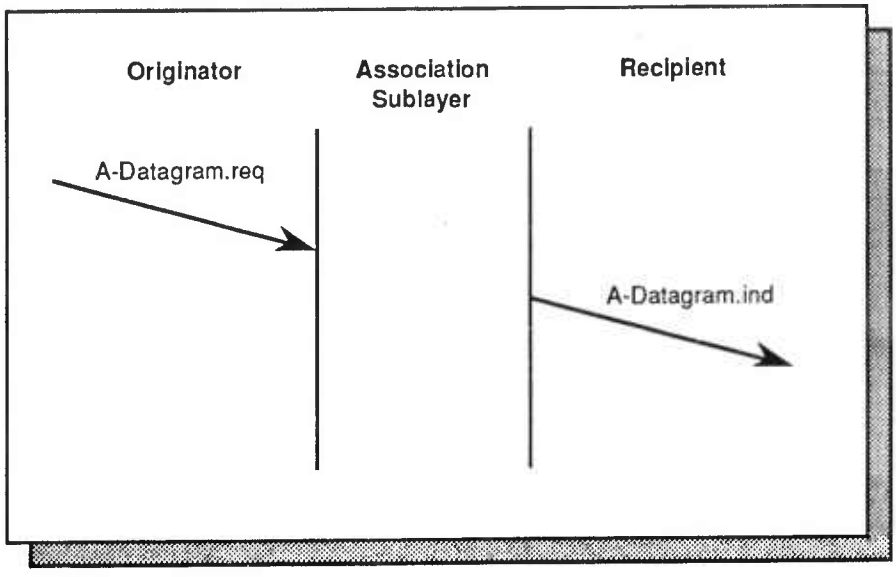


Figure 6–4. The sequence of service primitives associated with the datagram service.

The originator issues an A–Datagram request to the association sublayer to transfer a unit of data to one or more recipients. The association sublayer issues an A–Datagram indication to each recipient, conveying the unit of data.

The parameters associated with these primitives are:

Parameter Name	Req	Ind	Description
from–address	M	M(=)	the originator's address
multicast–group	M	M(=)	the group of recipients to which the unit of data is to be conveyed
protocol–identifier	M	M(=)	conveys from the originator to the recipients an identification of the protocol according to which the unit of data is to be interpreted
data	U	U(=)	the unit of data to be conveyed

The size of the unit of data to be conveyed must be a multiple of 8 octets.

6.4.3 Transaction service

The transaction service is used to transfer a unit of data from an originator to a designated recipient, and to return, in response, a unit of data from that recipient back to the originator. This is a "confirmed" service.

The recipient providing the response to the originator is termed the *respondent*. The originator specifies a multicast group of recipients, of which both the originator and respondent must be members. All other members of that group will receive both the originator's unit of data, and the unit of data returned by the respondent. These other multicast group members are called *observers*.

There are six primitives associated with this service:

A–Transact.req

A–Transact.ind

A–Transact.rsp

A–Transact.conf

A–Request.ind

A–Response.ind

The sequence in which these primitives are used is illustrated in figure 6–5.

The originator issues an A–Transact request to the association sublayer to transfer a unit of data to a respondent. The association sublayer issues an A–Transact indication to the respondent, and an A–Request indication to each observer, conveying the unit of data. The respondent issues an A–Transact response to the the association sublayer to return a unit of data to the originator. The association sublayer issues an A–Transact confirmation to the originator, and an A–Response indication to each observer, conveying the returned unit of data.

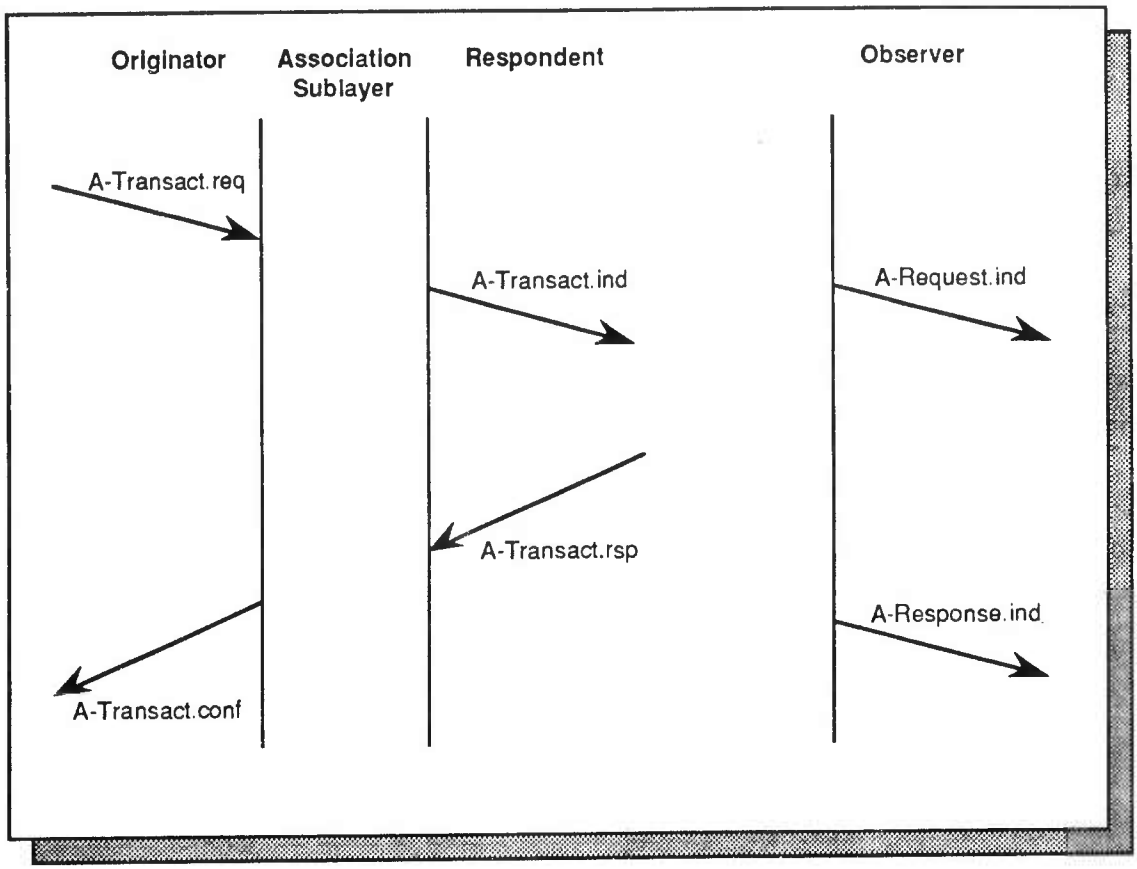


Figure 6–5. The sequence of service primitives associated with the transaction service.

The parameters associated with the A–Transact primitives are:

Parameter Name	Req	Ind	Rsp	Conf	Description
originator–address	M	M(=)			the originator's address
respondent–address	M	M(=)			the respondent's address
multicast–group	M	M(=)			the multicast group that includes the originator, the respondent, and all observers

Parameter Name	Req	Ind	Rsp	Conf	Description
protocol–identifier	M	M(=)			conveys from the originator to the recipients an identification of the protocol according to which both units of data is to be interpreted
request–data	U	U(=)			the unit of data to be conveyed from the originator to the respondent
response–data			U	U(=)	the unit of data to be conveyed from the respondent to the originator
cache–response			M		indicates whether the unit of data returned by the respondent should be cached by its association entity so that the entity can retransmit the response if necessary
result				M	indicates success or failure of the association sublayer operation

The originator's association entity can, if necessary, make several attempts to carry out the transaction. Each time, it will attempt to transmit the request data to the respondent's association entity, and, as a result, the respondent's association entity may receive the same request data multiple times. The cache–response parameter indicates how that association entity should respond in this situation. If the respondent has indicated that its response should be cached, then its association entity will return the same response data with each repeated attempt. If the response is not to be cached, the respondent's association entity will obtain a fresh copy of the response data with each repeated attempt. The cache–response parameter is available for the association sublayer user to indicate which behavior is appropriate in a particular situation.

The parameters associated with the A–Request indication have the same values as those supplied by the originator with its A–Transact request:

Parameter Name	Ind	Description
originator–address	M(=)	the originator's address
respondent–address	M(=)	the respondent's address
multicast–group	M(=)	the multicast group that includes the originator, the respondent, and all observers
protocol–identifier	M(=)	identifies the protocol according to which the unit of data is to be interpreted
request–data	U(=)	the unit of data to be conveyed from the originator to the respondent

The parameters associated with the A–Response indication have the same values as those supplied by the originator with its A–Transact request, and by the respondent with its A–Transact response:

Parameter Name	Ind	Description
originator–address	M(=)	the originator's address
respondent–address	M(=)	the respondent's address
multicast–group	M(=)	the multicast group that includes the originator, the respondent, and all observers
protocol–identifier	M(=)	identifies the protocol according to which the unit of data is to be interpreted
response–data	U(=)	the unit of data to be conveyed from the respondent to the originator

Each unit of data conveyed by the transaction service must be multiple of 8 octets in size.

6.5 Specification of the association protocol

We define the association protocol by first describing association protocol data units, and then describing the protocol procedures that use them.

Two protocol procedures comprise the association protocol: one implements the datagram service, the other, the transaction service. Each is independent of the other. No protocol procedure is invoked by use of the group subscription service, although each

association entity must maintain information about which multicast groups its user is currently subscribed to.

6.5.1 Association protocol data unit format

The formats of association protocol data units (APDUs) are defined here using the data representation notation documented in appendix A.

All APDUs have a length that is an integral multiple of 64 bits, and all begin with a common 64–bit header. Included in this header is a code indicating the kind of APDU present. Four kinds of APDUs are used by the association protocol:

- A Datagram APDU is used for the datagram service to convey a unit of data from the originator to the recipient(s).
- A Request APDU is used for the transaction service to convey a unit of data from the originator to the recipient(s).
- A Response APDU is used for the transaction service to convey a unit of data from the respondent to the originator and to other recipients.
- A Padding APDU is used to pad a datagram to some minimum length, as required by the network service.

APDUs are of the following form:

```
type AssociationPDU sequence (
    version             AssociationProtocolVersion,
    kind                AssociationPDUKind,
    dataLength          UnsignedInteger (8),
    group               MulticastGroupID,
    userProtocol        AssociationUserProtocol,
    originator          SimulationAddress,
    variant             choice (kind) of (

      when (datagramAPDUKind) datagram sequence (
        data            array (dataLength) of AssociationDataUnit
      ),

      when (requestAPDUKind) request sequence (
        respondent      SimulationAddress,
        transactionID   TransactionIdentifier,
                        unused (16),
        data            array (dataLength) of AssociationDataUnit
      ),

      when (responseAPDUKind) response sequence (
        respondent      SimulationAddress,
        transactionID   TransactionIdentifier,
                        unused (16),
        data            array (dataLength) of AssociationDataUnit
      ),

      when (paddingAPDUKind) padding sequence (
        data            array (dataLength) of AssociationDataUnit
      )
    )
)
```

The `version` field identifies the version of association protocol to which the APDU pertains. This allows new versions of the association protocol to be introduced without disruption to existing implementations. The association protocol described in this report has version number 2:

```
type AssociationProtocolVersion enum (4) {
    protocolVersionAug89 (1),
    protocolVersionJan90 (2)
}
```

The `kind` field identifies the kind APDU present:

```
type AssociationPDUKind enum (4) {
    datagramAPDUKind (1),
    requestAPDUKind (2),
    responseAPDUKind (3),
    paddingAPDUKind (4)
}
```

All four kinds of APDUs contain a `data` field whose length is an integral multiple of 64 bits. The `dataLength` field of the APDU header specifies the length of the data field, in multiples of 64 bits.

The `data` field contains the user's PDU. It is represented by data elements of the following form:

```
type AssociationDataUnit array(8) of UnsignedInteger (8)
```

Each APDU is directed to a particular multicast group, identified by the `group` field of the APDU header. The `userProtocol` field conveys a code describing the user protocol to which the `data` field content pertains.

```
type MulticastGroupID UnsignedInteger (8)

type AssociationUserProtocol UnsignedInteger (8)
```

A Datagram APDU is produced when an originator requests the datagram service of the association sublayer. The Request and Response APDUs are produced when an originator requests the transaction service. In all three cases, the `originator` field of the APDU header identifies the originator by its simulation address.

Request and Response APDUs include some additional fields required to implement the transaction service. The `respondent` field identifies the respondent as specified by the originator. The `transactionID` fields contain sequence numbers generated by the originator's association entity.

```
type TransactionIdentifier UnsignedInteger (16)
```

A Padding APDU may be inserted at the end of a network datagram in order to pad it to some minimum length, as required by the network service. The contents of the `data` field of a Padding APDU are not interpreted.

If an association entity receives from the network layer an APDU that is in error, it discards the APDU.

6.5.2 Datagram protocol procedure

The datagram protocol procedure implements the datagram service.

In response to an A–Datagram.req service primitive from its user, an application entity issues a Datagram APDU. The APDU contains the originator address, multicast group number, protocol identifier, and user data supplied by the user. It is delivered to all association entities by the network layer.

Upon receiving a correct Datagram APDU from the network layer, an association entity determines whether the APDU pertains to a multicast group to which its user has subscribed. If so, it issues an A–Datagram.ind service primitive to its user. The contents of the APDU are decoded to obtain the from–address, multicast–group, protocol–identifier, and data parameters of the service primitive. A Datagram APDU specifying a multicast group not subscribed to is discarded.

6.5.3 Transaction protocol procedure

The transaction protocol procedure implements the transaction service.

Figures 6–6 and 6–7 represent the behavior of the association entities representing the originator and the respondent. Initially, both entities are in their respective idle states.

For each transaction it initiates, the originator's association entity chooses a 16–bit value called a transaction identifier. Consecutive transaction identifiers are assigned to consecutive transactions; transaction identifier 65535 is followed by transaction identifier 0. The combination of the originator's simulation address and the transaction identifier serves to distinguish a transaction from among other transactions that may be at the same time.

The originator's association entity maintains the following state information for an active transaction during the transaction protocol procedure:

- The values of the parameters supplied with the A–Transact.req service primitive that initiated the transaction.

- The transaction identifier chosen for the transaction.
- A timer and a counter used to perform retries.

The recipient's association entity maintains the following state information for an active transaction during the transaction protocol procedure:

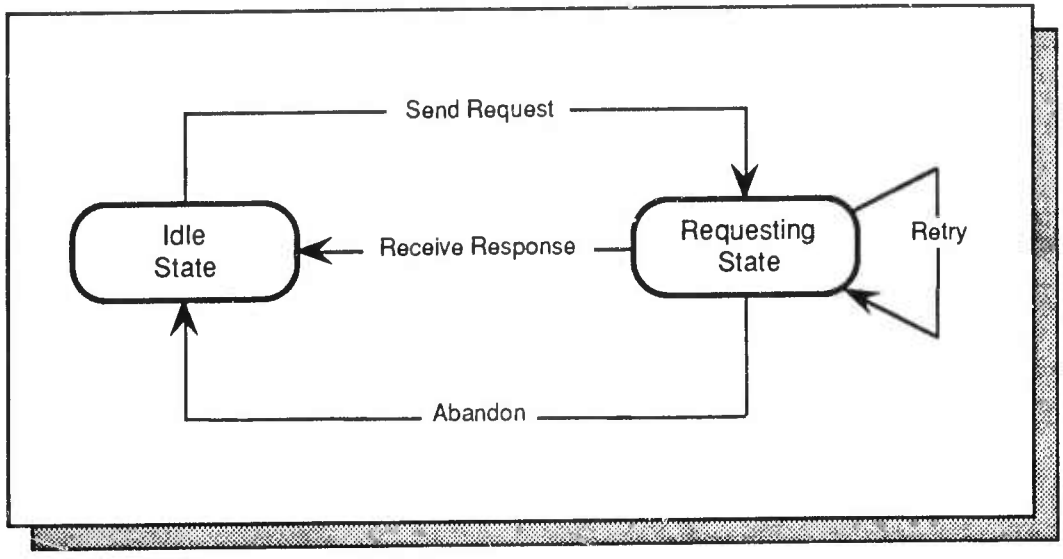
- The simulation address of the transaction's originator.
- The transaction identifier.
- If the user has specified that the response be cached, then the association entity maintains a timer and a copy of the response–data supplied by the user.

On receiving an A–Transact.req from its user, the originator's association entity prepares a Request APDU using parameters supplied by the user and a newly chosen transaction identifier. It then makes up to 3 attempts (`transactionRetryCount`) to send the Request APDU and receive a matching Response APDU. A received Response APDU matches the Request APDU if it bears the same originator and response simulation addresses, user protocol identifier, multicast group number, and transaction identifier. The retries are separated by a wait of 3 seconds (`transactionRetryTime`). Upon receiving a matching Response APDU, the association entity issues an A–Transact.conf to its user indicating success. Upon timing out after the last retry, it issues one indicating failure.

On receiving a Request APDU from the network service, an association entity determines whether it's own user subscribes to the multicast group under which the Request APDU was issued. If not, the Request APDU is discarded. Otherwise, the association entity then determines whether its own user is identified as the transaction's respondent. If not, it issues an A–Request.ind to its user with parameters obtained from the Request APDU.

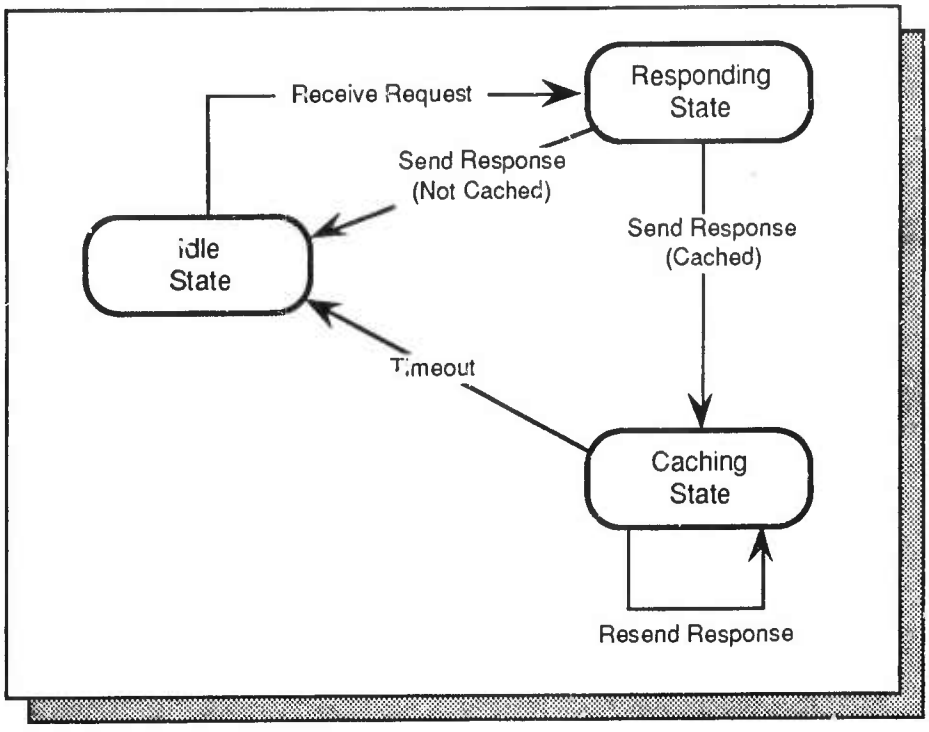
If an association entity receives a Request APDU with an appropriate multicast group number and its user identified as the respondent, it determines whether it has cached a response matching the request. If so, it creates a Response APDU from the cached information and reissues it to the network service. Otherwise, it issues an A–Transact.ind to its user (the respondent) with parameters obtained from the Request APDU. From the A–Transact.rsp returned by the respondent, the association entity creates a Response APDU, which it issues to the network service. If the respondent specifies that the response is to be cached, the association entity retains the response and enters its caching

state for a period of 10 seconds (`transactionCacheTime`) before returning to idle state. Otherwise, when the response is not to be cached, it returns directly to idle state.



Transition	Condition and Action
Send Request	When an A–Transact.req service primitive is received from the user, create a Request APDU and send it to all association entities using the network service. Set a timer to 3 seconds (`transactionRetryTime`) and a counter to 3 (`transactionRetryCount`).
Receive Response	When a Response APDU related to the transaction is received from the network service, issue an A–Transact.conf service primitive to the user indicating that the transaction has succeeded. Cancel the timer.
Retry	When the timer expires, decrement the counter. If it is nonzero, resend the Request APDU and reset the timer to 3 seconds (`transactionRetryTime`).
Abandon	When the timer expires, decrement the counter. If it is zero, issue an A–Transact.conf service primitive to the user indicating that the transaction has failed.

Figure 6–6. Behavior of the association entity serving the originator of a transaction.



Transition	Condition and Action
Receive Request	When a Request APDU identifying the user as respondent is received from the network service, and the request does not match a response already cached, issue an A–Transact.ind service primitive to the user.
Send Response	When an A–Transact.rsp service primitive is received from the user, create a Response APDU and send it to all association entities using the network service. If response caching is specified, set a timer to 10 seconds (`transactionCacheTime`) and enter caching state. Otherwise, enter idle state.
Resend Response	When a Request APDU identifying the user as respondent is received from the network service, and the request matches a response already cached, resend the Response APDU.
Timeout	When the timer expires, discard the cached response.

Figure 6–7. Behavior of the association entity serving the respondent of a transaction.

On receiving a Response APDU from the network service, an association entity determines whether the APDU pertains to an active transaction for which the entity is representing the originator. If so, the procedure is as described above. If not, the entity determines whether its own user subscribes to the multicast group under which the

Response APDU was issued. If not, the Response APDU is discarded. Otherwise, the entity issues an A–Response.ind to its user with parameters obtained from the Response APDU.

7 SIMULATION PROTOCOL

The simulation protocol is used by simulators to communicate with each other information about the simulated world. The protocol serves to initiate vehicles into an exercise, withdraw them from an exercise, describe the externally visible appearance of vehicles, report the firing and impact of projectiles, transfer supplies between vehicles, and effect repairs to vehicles.

7.1 Simulation protocol data units

The simulation protocol makes use of several kinds of protocol data units. All simulation PDUs have a length that is an integral multiple of 64 bits. In some cases, this may require padding to be included at the end of the PDU. All simulation PDUs begin with a common 64–bit header. Included in this header is a code indicating the kind of PDU present:

```
type SimulationPDUKind enum (8) {
    activateRequestPDUKind (1),          -- Activate Request PDU
    activateResponsePDUKind (2),         -- Activate Response PDU
    deactivateRequestPDUKind (3),        -- Deactivate Request PDU
    deactivateResponsePDUKind (4),       -- Deactivate Response PDU
    vehicleAppearancePDUKind (5),        -- Vehicle Appearance PDU
    radiatePDUKind (6),                  -- Radiate PDU
    firePDUKind (7),                     -- Fire PDU
    impactPDUKind (8),                   -- Impact PDU
    indirectFirePDUKind (9),             -- Indirect Fire PDU
    collisionPDUKind (10),               -- Collision PDU
    serviceRequestPDUKind (11),          -- Service Request PDU
    resupplyOfferPDUKind (12),           -- Resupply Offer PDU
    resupplyReceivedPDUKind (13),        -- Resupply Received PDU
    resupplyCancelPDUKind (14),          -- Resupply Cancel PDU
    repairRequestPDUKind (15),           -- Repair Request PDU
    repairResponsePDUKind (16)           -- Repair Response PDU
    markerPDUKind (17),                  -- Marker PDU
    breachedLanePDUKind (18),            -- Breached Lane PDU
    minefieldPDUKind (19),               -- MineField PDU
}
```

PDUs containing an unknown `kind` field should be ignored. Kind values in the range of 129 to 255 are reserved for temporary or experimental use.

Following the PDU header is a portion whose format depends on the kind of PDU. The overall content of a PDU is:

```
type SimulationPDU sequence {
    version             SimulationProtocolVersion,
    kind                SimulationPDUKind,
    exercise            ExerciseID,
                        unused (40),
    variant             choice (kind) of {

      when (activateRequestPDUKind)
        activateReq     ActivateRequestVariant,

      when (activateResponsePDUKind)
        activateRsp     ActivateResponseVariant,

      when (deactivateRequestPDUKind)
        deactivateReq   DeactivateRequestVariant,

      when (deactivateResponsePDUKind)
        deactivateRsp   DeactivateResponseVariant,

      when (vehicleAppearancePDUKind)
        appearance      VehicleAppearanceVariant,

      when (radiatePDUKind)
        radiate         RadiateVariant,

      when (firePDUKind)
        fire            FireVariant,

      when (impactPDUKind)
        impact          ImpactVariant,

      when (indirectFirePDUKind)
        indirectFire    IndirectFireVariant,

      when (collisionPDUKind)
        collision       CollisionVariant,

      when (serviceRequestPDUKind)
        serviceReq      ResupplyVariant,

      when (resupplyOfferPDUKind)
        resupplyOffer   ResupplyVariant,

      when (resupplyReceivedPDUKind)
        resupplyReceived ResupplyVariant,
```

```
        when (resupplyCancelPDUKind)
          resupplyCancel  ResupplyCancelVariant,

        when (repairRequestPDUKind)
          repairReq       RepairRequestVariant,

        when (repairResponsePDUKind,
          repairRsp       RepairResponseVariant

        when (markerPDUKind)
          marker          MarkerVariant,

        when (breachedLanePDUKind)
          breachedLane    BreachedLaneVariant,

        when (minefieldPDUKind)
          minefield       MinefieldVariant

    }
)
```

The `version` field specifies the version of simulation protocol to which the PDU pertains. The use of this field allows new versions of the simulation protocol to be introduced without disruption to existing implementations. The simulation protocol described in this report has version number 3:

```
type SimulationProtocolVersion enum (8) {
    simProtocolVersionAug89 (1)
    simProtocolVersionJan90 (2),
    simProtocolVersionJan90Corrected (3)
)
```

The `exerciseID` field identifies the exercise to which the PDU pertains (§5.1.6).

7.2 Use of association sublayer services

Simulation PDUs are conveyed among simulators using the services of the association sublayer defined in chapter 6. A single PDU is issued through a single invocation of either the A–Datagram.req service primitive or the A–Transact.req service primitive. The discussion of protocol procedures, below, specifies which of these two service primitives is used in each case.

To distinguish the simulation protocol from other protocols using the association sublayer, the simulation protocol is assigned a unique association sublayer user protocol number. This number is 1:

```
constant simulationProtocolNumber 1
```

Every simulation protocol interaction among simulation entities takes place within the context of a particular simulation exercise. Associated with an exercise is an exercise identifier, which distinguishes it from other, concurrent exercises. With but one exception noted below, the interactions associated with a particular exercise are carried by the association service using a multicast group number that is identical to the exercise's identifier. This allows simulation entities to receive information only about the exercises of interest to them by subscribing only to selected multicast groups.

The one exception to the rule stated above on the use of exercise identifiers as multicast group numbers occurs with the protocol interaction used to introduce a simulation entity into an exercise. That protocol interaction is performed using the multicast group number 0 (a group that includes all simulation entities) to ensure that a simulation entity may be communicated with regardless of which multicast groups it has already subscribed to.

Some protocol interactions call for using the transaction service to issue a PDU to the simulation entity that is modeling a particular vehicle, *x*. Use of the transaction service requires knowing the simulation address of the respondent—in this case, the address of the simulation entity modeling *x*. That address is discovered through the process of receiving PDUs that describe *x*: the A–Datagram.ind service primitives that deliver these PDUs also specify the simulation address of their originator. This simulation address should be used for directing transactions to the simulation entity modeling *x*.

An alternate method of obtaining the address of a vehicle's simulation entity, by extracting the `simulation` component of its vehicle identifier (§5.1.24), must not be relied upon since it precludes transferring the simulation of a vehicle from one entity to another.

The transaction service of the association sublayer allows a respondent to specify whether a particular response should be cached for retransmission. This service element is chosen using the cache–response parameter of the A–Transact.rsp service primitive. Use of this feature for simulation protocol interactions is optional in all cases.

7.3 Protocol procedures

The simulation protocol is logically divided into several distinct protocol procedures, each of which provides a related set of functions. These procedures are defined individually in the following subsections.

All protocol procedures involve the exchange of PDUs among simulation entities using services of the association sublayer. For brevity, we sometimes use the term *simulator* in this section when referring to simulation entities. When describing use of the association sublayer, however, we generally revert to the more correct term, *simulation entity*. Moreover, we usually shorten "the transaction service of the association sublayer" to "the underlying transaction service".

7.3.1 Activation

One simulator may prompt another to begin simulating a vehicle through a procedure called *activation*.

The activation procedure is commonly used by MCC systems to start crewed vehicle simulators such as the M1 Abrams main battle tank simulator. An active vehicle can also be re–activated through this same process, in effect resetting the simulation of that vehicle. Activation is used for this purpose when an MCC system that has simulated the towing (recovery) of a vehicle repositions that vehicle on the terrain at its towing destination.

The simulation entity requesting the activation uses the underlying transaction service to convey an Activate Request PDU to the entity that is to perform the simulation. The Activate Request PDU includes information describing the vehicle to be simulated. An Activate Response PDU is then returned to indicate whether the activation request has been accepted.

This transaction is performed using the association sublayer's multicast group 0 (the broadcast group) to ensure that the transaction will be received by the respondent regardless of which multicast groups it has subscribed to.

An Activate Request PDU includes the following fields in addition to its PDU header:

```
type ActivateRequestVariant sequence {

    -- Purpose of the activation:
    reason              ActivateReason,

    -- Identity of the activated vehicle:
    vehicleClass        VehicleClass,
    vehicleID           VehicleID,
    unit                OrganizationalUnit,
    marking             VehicleMarking,
    guises              VehicleGuises,

    -- Information about the simulated world:
    simulatedTime       Time,
    terrain             TerrainDatabaseID,
    battleScheme        BattleScheme,

    -- Status of the vehicle:
    onSurface           Boolean,
                        unused (23),
    status              VehicleStatus,
    location            WorldCoordinates,

    -- Depending on vehicle class:
    specific            choice (vehicleClass) of {

      -- A static vehicle:
      when (vehicleClassStatic) static sequence {
        hullAzimuth     Angle,
                        unused (32)
      },

      -- A simple moving vehicle, without a turret:
      when (vehicleClassSimple) simple sequence {
        hullAzimuth     Angle,
                        unused (32)
      },

      -- A tank:
      when (vehicleClassTank) tank sequence {
        hullAzimuth     Angle,
        turretAzimuth   Angle
      }
    },
```

```
    -- These fields are optional:
        -- Initial velocity and freeze state
        velocity            VelocityVector,
        freezeState         Boolean,
                            unused (31),

        -- More information about the simulated world:
        VLVisibility        Float (32), -- visibility in visible light in meters
        simulatedSkyColor   SkyColor,
                            unused (24)
}
```

The `reason` field indicates the purpose of the activation:

```
type ActivateReason enum (8) {
    activateReasonOther,        -- none of those listed below
    exerciseStart,              -- initial entry into new exercise
    exerciseRestart,            -- restart of exercise
    vehicleReconstitution,      -- restoration of vehicle status
    towingArrival               -- towing destination reached
}
```

When a vehicle is first introduced into a new exercise, the activation reason is specified as `exerciseStart`. If the exercise is being resumed after some interruption, a reason of `exerciseRestart` is specified. If the vehicle is being activated to provide it a new location, operational status, or supply of munitions, a reason of `vehicleReconstitution` is specified. In all three cases, and in the case where the reason is specified as `activateReasonOther`, the remaining fields of the Activate Request PDU are interpreted as completely specifying the new state of the activated vehicle.

After a vehicle has been towed to a new destination, and upon reaching that destination, it may be activated using an Activate Request PDU in which the `reason` field is specified as `towingArrival`. In this case, only the `vehicleClass`, `location`, `onSurface`, and `specific` fields are interpreted as specifying a new location and orientation for the vehicle. The vehicle will retain other state attributes it had prior to towing.

The `vehicleID` field is relevant when the reason for the activation is `exerciseRestart`, `vehicleReconstitution`, or `towingArrival`. In these cases, the field identifies the particular vehicle being (re)activated by referring to a vehicle that has already been participating in the exercise (§5.1.24). In all other cases, the

vehicleID field contains zero, and the activated simulator is responsible for selecting a new, unique identifier for the activated vehicle.

The organizational unit to which the vehicle is assigned it described by the unit field (§5.1.11). The marking field describes a label on the vehicle, such as a bumper number, that may be visible from other vehicles under certain circumstances (§5.1.25).

The guises field supplies two object type codes, specifying how the activated vehicle is to appear when viewed from other vehicles (§5.1.23). Both object type codes are to be included in the Vehicle Appearance PDUs subsequently issued for the vehicle.

The simulatedTime field specifies the simulated time at which the PDU is issued (§5.1.18). Its purpose is to provide the receiving simulator with information necessary for modeling a particular time of day or time of year. Because the PDU may be subject to delay and retransmission by the association service and its underlying network, the time it carries should be interpreted as being accurate to at most ±10 seconds.

The terrain field identifies the terrain database to be used by the activated simulator (§5.1.17).

The vehicle's status—the health of its various subsystems and the quantities of supplies it has on board—will be as described by the status field (§5.1.26). For a vehicle newly introduced into an exercise, the odometer component of the status field will be zero; for one that is being re-activated, perhaps as an exercise is being restarted, the odometer component will reflect the distance travelled earlier by that vehicle in the same exercise.

The onSurface and location fields provide the activated vehicle's initial location. If onSurface is false, the vehicle will exist at the location in space specified by the world coordinates (§5.1.29). Otherwise, when onSurface is true, the vehicle will exist upon the terrain's surface at the location specified by the x and y components of the world coordinates. (In this case, the z component is ignored.)

The latter portion of the PDU contains vehicle-specific information whose format varies with the class of vehicle being activated. The correct interpretation of this portion is determined by the contents of the PDU's vehicleClass field (§5.1.20). The direction the vehicle is to face initially is represented by the hullAzimuth field as an angle measured counterclockwise from north (i.e., a value of 2^{30} (90 degrees) means facing

west). If the vehicle is of the tank class, the azimuth of its turret relative to its hull is represented by the `turretAzimuth` field, which measures the angle counterclockwise from the front of the hull (§5.1.1).

The final portion of the PDU contains optional fields. The presence of these fields may be determined by the length returned by the association protocol. As with any PDU, a simulator that receives an Activate Request PDU that includes unknown fields at the end should ignore these unknown fields.

The first set of optional fields supports the activation of a simulator with an initial velocity and freeze state. These fields may be particularly useful for initializing aircraft simulators. The `velocity` field specifies the vehicle's initial velocity vector (§5.1.28). If `freezeState` field is true, the vehicle should be initialized in a "frozen" or suspended state.

The second set of optional fields supports the initialization of visibility and sky state. It may only be present if the fist set of optional fields is also present. The `VLVisibility` field specifies the visible light visibility in meters. The `simulatedSkyColor` field is represented by SkyColor data element which is defined as follows:

```
type SkyColor enum (8) {
    skyColorClear,          -- No cloud cover
    skyColorPartlyCloudy,   -- 0-50% cloud cover
    skyColorPartlySunny,    -- 50-100% cloud cover
    skyColorOvercast,       -- total, light colored cloud cover
    skyColorRainy           -- total, dark colored cloud cover
}
```

A simulator that correctly receives an Activate Request PDU must immediately respond by returning an Activate Response PDU. This PDU is simply an acknowledgement of receipt; it does not represent that the simulator has successfully completed the activation process.

An Activate Response PDU includes the following fields in addition to its PDU header:

```
type ActivateResponseVariant sequence {
    vehicleID           VehicleID,
    result              ActivateResult,
                        unused (8),
    timeLimit           UnsignedInteger (16),
                        unused (48)
}

type ActivateResult enum (8) {
    activateRequestAccepted,
    invalidActivateParameter,
    unexpectedActivateReason,
    invalidVehicleIdentifier,
    terrainDatabaseUnavailable
}
```

The `result` field indicates whether the activation request has been accepted, and, if not, why not.

If the activation request has been accepted, then the `vehicleID` field contains the vehicle identifier of the activated vehicle (§5.1.24). Otherwise, it contains zeros.

The responding simulator may require some period of time before it is able to issue Vehicle Appearance PDUs for the newly activated vehicle. This will be the case, for example, if the simulator must perform considerable processing in order to initialize itself. If the simulator accepts the activation request, and therefore returns a result of `activateRequestAccepted`, then it must specify an upper limit for this period of time. The limit is specified in the `timeLimit` field, in units of seconds. If the simulator does not accept the activation request, and therefore returns a result other than `activateRequestAccepted`, then the `timeLimit` field should contain 0.

7.3.2 Deactivation

A simulator may withdraw its own vehicle from an exercise at any time, or it may be requested by another simulator (such as an MCC system) to withdraw it through a process called *deactivation.* In either case, the withdrawal of the vehicle is announced using a Deactivate Request PDU. This PDU is used in two ways:

- If a simulation entity is withdrawing its own vehicle, it will eonvey the Deaetivate Request PDU to all other exercise participants using the underlying datagram service. Those receiving the Deaetivate Request PDU then eease to dead reckon and display the withdrawn vehicle.

- If one simulation entity is requesting that another cease simulating a vehicle, it will convey the Deactivate Request PDU to the simulating entity using the underlying transaction service. That entity responds with a Deactivate Response PDU, and ceases simulating its vehicle. Other simulation entities receiving the same Deactivate Request PDU simply cease to dead reckon and display the deactivated vehicle.

In both cases, a multicast group number identical to the exercise identifier is used to invoke the association service.

A Deactivate Request PDU includes the following fields in addition to its PDU header:

```
type DeactivateRequestVariant sequence {
    vehicleID           VehicleID,
    reason              DeactivateReason,
                        unused (8)
}
```

The `vehicleID` field contains the identifier of the vehicle withdrawing (§5.1.24).

The `reason` field indicates the purpose of the deactivation:

```
type DeactivateReason enum (8) {
    deactivateReasonOther,          -- none of those listed below
    exerciseEnd,                    -- end of exercise
    vehicleWithdrawn,               -- vehicle withdrawn from exercise
    vehicleDestroyed,               -- vehicle no longer exists
    towingDeparture                 -- start of towing operation
}
```

A Deactivate Response PDU includes the following fields in addition to its PDU header:

```
type DeactivateResponseVariant sequence {
    vehicleID           VehicleID,
    result              DeactivateResult,
                        unused (8)
}

type DeactivateResult enum (8) {
    deactivateRequestAccepted,
    invalidDeactivateParameter,
    unexpectedDeactivateReason,
    vehicleNotActive
}
```

The `vehicleID` field is identical to that in the corresponding Deactivate Request PDU. The `result` field indicates whether the deactivation request has been accepted, and, if not, why not.

7.3.3 Appearance and other state updates

State updates are issued periodically by simulators to describe some element of the state of the system they simulate. A simulator learns of the existence of another simulated systems when it first receives on of these updates, which may be at any point during an exercise. State updates are used to describe vehicles, minefields, and radar emissions. When an update for a given system has not been received for an defined interval of time, the system is no longer considered to exist (or in the case of radar, to no longer be emitting).

Vehicle Appearance PDU

A simulator periodically reports information about a vehicle it simulates so that other simulators may correctly depict that vehicle. Information about the visual appearance of a vehicle is issued as a Vehicle Appearance PDU describing the vehicle at the moment of issue. For some types of vehicles, this PDU contains additional information used by other simulators to dead reckon the vehicle's appearance from that moment forward.

The underlying datagram service is used to issue a Vehicle Appearance PDU.

A simulator will issue a new Vehicle Appearance PDU for its vehicle whenever the discrepancy between the vehicle's actual appearance and its dead reckoned appearance exceeds one of the discrepancy thresholds defined in chapter 3. It will also issue a new Vehicle Appearance PDU if 5 seconds (`vehicleAppearanceTime`) have elapsed since it issued the last one, to ensure that new simulators joining an exercise will promptly learn of all existing vehicles.

If a simulator ceases to receive Vehicle Appearance PDUs describing a particular vehicle, and none are received for a period of 12 seconds (`vehicleDisappearanceTime`), hen the simulator may assume that that vehicle no longer exists. (A Deactivate Request)U also serves to indicate that a vehicle no longer exists.)

A Vehicle Appearance PDU includes the following fields in addition to its PDU header:

```
type VehicleAppearanceVariant sequence {

    -- Identity of the vehicle:
    vehicleID           VehicleID,
    vehicleClass        VehicleClass,
    force               ForceID,

    -- Appearance of the vehicle:
    guises              VehicleGuises,
    location            WorldCoordinates,
    rotation            array (3,3) of Float (32),
    appearance          UnsignedInteger (32),
    marking             VehicleMarking,
    timestamp           UnsignedInteger (32),
    capabilities        VehicleCapabilities,
    engineSpeed         UnsignedInteger (16),
    stationary          Boolean,
                        unused (7),

    -- Reason for issuing the PDU:
    reason              AppearanceUpdateReason,

    -- Depending on vehicle class:
    specific            choice (vehicleClass) of {

      -- A simple moving vehicle, without a turret:
      when (vehicleClassSimple) simple sequence {
        velocity        VelocityVector,
                        unused (32)
      },

      -- A tank:
      when (vehicleClassTank) tank sequence {
        velocity        VelocityVector,
        turretAzimuth   Angle,
        gunElevation    Angle,
                        unused (32)
      }
    }
}
```

The `vehicleID` field identifies the vehicle described by the Vehicle Appearance PDU (§5.1.24). The `force` field identifies the force to which it has been assigned (§5.1.7).

The `guises` field contains two object type codes describing the appearance of the vehicle as viewed from other vehicles (§5.1.23). Which of the two codes applies depends on what force the observing vehicle has been assigned to. The dependency is explained in section 2.3.

The vehicle's position and orientation in the world coordinate system are described by the `location` and `rotation` fields. The `location` field contains the position, in world coordinates, of the vehicle's own coordinate system origin (§5.1.29). The elements of the `rotation` field, which is a nine element rotation matrix as defined in section 2.4, appear in the order r_{11}, r_{12}, r_{13}, r_{21}, ..., r_{33} (i.e., in row–major order).

The `appearance` field contains 32 bits describing modifications to the vehicle's basic appearance. The convention for assigning bits within this field is to use the low–order 16 bits for vehicle appearance attributes that may apply to many types of vehicle, and to use the high–order 16 bits for attributes that are specific to certain types of vehicles.[7] The bits within this field are presently used as follows (with bit 0 being the least–significant, or rightmost, bit):

Name	Bits	Purpose
`vehDestroyed`	0	is 1 if the vehicle is destroyed, and 0 otherwise
`vehSmokePlume`	1	is 1 if a plume of smoke is rising from the vehicle, and 0 otherwise
`vehFlaming`	2	is 1 if flames are rising from the vehicle, and 0 otherwise
`vehDustCloudMask`	3 – 4	describes any dust cloud being raised by the vehicle: 0: no dust cloud 1: small dust cloud 2: medium dust cloud 3: large dust cloud
`vehMobilityDisabled`	5	is 1 if the vehicle appears unable to move

[7] If many additional appearance modifier bits are required for newly defined types of vehicles, it is expected that certain bits will have to bear different meanings according to what type of vehicle they describe.

vehFirepowerDisabled	6	is 1 if the vehicle appears unable to shoot
vehCommunicationDisabled	7	is 1 if the vehicle appears unable to communicate
vehShaded	8	is 1 if the vehicle is in shadow
vehTOWLauncherUp	30	is 1 if the vehicle is an M2 or M3 with its TOW missile launcher raised, and 0 otherwise
vehEngineSmoke	31	is 1 if the vehicle is a T72 emitting engine smoke, and 0 otherwise
lfPositionMask	30-31	represents a soldier stance, if the "vehicle" is dismounted infantry 0: unknown 1: standing 2: kneeling 3: lprone

All other bits of the appearance field should remain zero.

The marking field describes a vehicle label, such as a bumper number, that may be visible on the vehicle for some observers (§5.1.25).

The timestamp field allows a determination of the relative timing of consecutive Vehicle Appearance PDUs describing the same vehicle. The value of this field is such that, for any two consecutive Vehicle Appearance PDUs describing the vehicle at times x milliseconds apart, the timestamp field of the latter Vehicle Appearance PDU is greater than that of the earlier by the amount x. In the first Vehicle Appearance PDU produced for a vehicle in an exercise, the timestamp field may have an arbitrary value. Section 3.4 explains how the timestamp may be used by a receiving simulator.

The capabilities field describes the vehicle's capabilities for resupplying, recovering, and repairing other vehicles (§5.1.19).

The `engineSpeed` field contains the vehicle's engine speed, in revolutions per seconds. It is present in the Vehicle Appearance PDU to allow simulators to synthesize the sounds produced by nearby vehicles.[8]

The `stationary` field, a Boolean, is true if the vehicle's velocity is zero, and false otherwise.

The format of the remaining portion of the Vehicle Appearance PDU depends on the class of vehicle it describes, as represented by the `vehicleClass` field (§5.1.20). For a vehicle of the static class, there are no additional fields. Vehicles of the simple and tank classes are further described by their velocity vectors, expressed relative to the world coordinate system in meters per second (§5.1.28). For tank class vehicles, turret azimuth and gun elevation angles are also present (§5.1.1). The turret azimuth is zero when the turret is aligned with the front of the tank, and it increases as the turret rotates counterclockwise (as viewed when looking down on the tank). The gun elevation is zero when the gun is parallel with the tank chassis, and it increases as the gun elevates.

A series of consecutive Vehicle Appearance PDUs describes a single vehicle at consecutive points in time, for the period of time that that vehicle is active. The series ends when a Deactivate PDU is issued for the vehicle, or when no Vehicle Appearance PDUs have been issued for 12 seconds (`vehicleDisappearanceTime`). Only certain Vehicle Appearance PDU fields are permitted to change from one PDU in the series to the next. The `appearance`, `rotation`, `timestamp`, `engineSpeed`, and `specific` fields may change from one PDU to the next. The `location` and `stationary` fields may only change from one PDU to the next if the vehicle is of the simple class or of the tank class. The `vehicleID`, `vehicleClass`, `force`, `guises`, `marking`, and `capabilities` fields do not change from one PDU to the next within the same series. However, these fields may change between one series and the next—i.e., between two series of Vehicle Appearance PDUs that refer to the same vehicle identifier, but that are separated by a Deactivate PDU or by a silence of at least 12 seconds (`vehicleDisappearanceTime`).

---

[8] At present no discrepancy thresholds are applied to the engine speed attribute of a vehicle's "appearance". Hence, changes in engine speed alone do not result in the issuance of Vehicle Appearance PDUs.

Radar

A Radiate PDU is periodically issued by the simulator of a vehicle possessing a radar. It reports the set of target vehicles illuminated by the radar—allowing the implementation of radar warning receivers—and it identifies the subset of those targets that were actually detected by the radar to aid analysis of an exercise. The Radiate PDU also describes the location of the radar emitter, and certain characteristics of its signal.

The rate at which Radiate PDUs are issued depends on the type of radar simulated, and the number of targets illuminated by the radar. If a search radar is simulated, it is expected that each scan of the radar will result in the issue of a single Radiate PDU—or several PDUs if all targets cannot be described in a single PDU. If a tracking radar is simulated, the simulator will issue Radiate PDUs at a fixed rate as long as the radar continues to track a target; this rate is not currently specified.

A Radiate PDU is issued using the underlying datagram service. In addition to its PDU header, it includes the following fields:

```
type RadiateVariant sequence {
    vehicleID           VehicleID,
    mode                RadarMode,
    dutyCycle           RadarDutyCycle,
    location            WorldCoordinates,
    carrierFrequency    SignalFrequency,
    signalPower         SignalPower,
    antennaGain         Float (32),
    pulseEnergy         Float (32),             -- joules
    numberIllumed       UnsignedInteger (8),
    numberDetected      UnsignedInteger (8),
    targetID            array (numberIllumed) of VehicleID
}
```

The `vehicleID` field includes the identifier of the vehicle containing the radar (§5.1.24).

Two fields, `mode` and `dutyCycle`, characterize the type of radar:

```
type RadarMode enum (8) {
    radarModeOther,                    -- none of those listed below
    radarModeSearch,
    radarModeAcquisition,
    radarModeTracking
}

type RadarDutyCycle enum(8) {
    radarDutyCycleOther,               -- none of those listed below
    radarDutyCycleContinuous,
    radarDutyCyclePulsed
}
```

The location of the radar emitter, in world coordinates, is supplied in the `location` field (§5.1.29).

The `carrierFrequency` , `signalPower`, `antennaGain`, and `pulseEnergy` fields describe the radiated signal. The carrier frequency is specified in hertz:

```
type SignalFrequency Float (32)        -- hertz
```

The power and energy of the radiated signal are characterized by three values. The power input to the antenna is specified in watts by the `signalPower` field:

```
type SignalPower Float (32)            -- watts
```

The antenna's efficiency is specified by the `antennaGain` field, as a fraction. The energy of a radar pulse is specified by the `pulseEnergy` field, in joules.

The `targetID` field contains from 0 to 33 identifiers of vehicles illuminated by the radar (§5.1.24).

```
constant maxRadiateTargets 33
```

The exact number of vehicle identifiers present is specified by the value of the `numberIllumed` field. The vehicle identifiers are ordered so that those vehicles actually detected by the radar are placed at the front of the list, and the `numberDetected` field specifies how many targets there are in this first part of the list. Of course, the value of the `numberDetected` field will never exceed the value of the `numberIllumed` field.

If necessary, the `targetID` field is followed by 32 unused bits so that the Radiate PDU's overall size is a multiple of 64 bits.

Mine fields

The SIMNET simulated battlefield supports mine fields. These minefields may be created, cleared, and marked with flags. Three distinct PDUs are used to support minefields: the Mine Field PDU, Breached Lane PDU, and the Marker PDU. Mine fields represent the actual mine fields. They are not necessarily detectable without special equipment. Breached lanes represent areas from which mines have been cleared. Mine field markers are flags that may be used by troops to mark the boundaries of a mine field. They are visible to the crews of simulators.

Mine Field PDU

The Mine Field PDU describes an emplaced mine field. A simulator receiving a Mine Field PDU may use the information to depict the mine field. A Mine Field PDU is issued whenever a mine field is created and once every 30 seconds (`mineFieldTime`) thereafter A simulator may assume a mine field no longer exists if it has not received any Mine Field PDUs from that particular mine field for a period of 66 seconds (`mineFieldTimeOut`).

The Mine Field PDU is issued using the underlying datagram service .

A Mine Field PDU includes the following fields in addition to its PDU header:

```
type MinefieldVariant sequence {
    mines               array (maxMineDescriptors) of MineDescriptor,
    vertices            array (maxMinefieldVertices) of XYCoordinates,
    numberOfVertices    UnsignedInteger (8),
    numberOfMineTypes   UnsignedInteger (8),
    minefieldID         ObjectID,
    emplacementTime     Time,  -- When the minefield was placed
    force               ForceID,
                        unused (56)
}
```

The `mines` field identifies the type and density of mines that make up the mine field. Each type of mine in the mine field is described by a `MineDescriptor` data element of the following form:

```
type MineDescriptor sequence {
    mineType            ObjectType,
    density             Float (32)
}
```

The `mineType` field identifies the type of mine (§5.1.10) The `density` field identifies the areal density of the `mineType` type of mine in the mine field in units of mines per square meter.

A maximum of 3 different types of mines may exist in a single mine field:

```
constant maxMineDescriptors 3
```

The `vertices` field describes the geometry of the mine field. The mine field is represented as a two-dimensional polygon. Each vertex is described by an XYCoordinates data element (§5.1.30). The polygon representing the mine field does not cross itself and no vertex is repeated. The mine field's location in the simulated world is determined by the projection of the specified polygon onto the surface of the terrain along the Z axis of the world coordinate system (§2.4).

The `numberOfVertices` field specifies the number of vertices actually used to describe the polygon. The polygon may have a maximum of thirteen vertices:

```
constant maxMinefieldVertices 13
```

The `numberOfMineTypes` field specifies the number of different mine types actually described by the `mines` field.

The `minefieldID` field uniquely identifies the mine field described by the Mine Field PDU (§5.1.9).

The `emplacementTime` field indicates the time the mine field was originally emplaced(§5.1.18). This time corresponds to the time the first Mine Field PDU was issued for this mine field. The emplacement time is required to interpret the effect of multiple overlapping mine fields and breached lanes.

The `force` field identifies the force which originally emplaced the mine field (§5.1.7).

Breached Lane PDU

The Breached Lane PDU describes a path that has been cleared through a mine field. It may be used by simulators capable of sensing mines to indicate this cleared path. A Breached Lane PDU is issued whenever a breached lane is created and once every 30 seconds (`mineFieldTime`) thereafter. A simulator may assume a breached lane no longer exists if it has not received any Breached Lane PDUs from that particular breached lane for a period of 66 seconds (`mineFieldTimeOut`).

The Breached Lane PDU is issued using the underlying datagram service. In addition to its PDU header, it includes the following fields:

```
type BreachedLaneVariant sequence {
    vertices            array (4) of XYCoordinates,  -- a "lane"
    breachedLaneID      ObjectID,
    numberOfVertices    UnsignedInteger (8),
    force               ForceID, -- the force of the breachers
    breachTime          Time  -- when the lane was breached
}
```

The `vertices` field specifies the geometry of the breached lane as follows: The vertices define a three or four sided polygon in the X-Y plane of the world coordinate system (§5.1.29). The location of the breached lane in the simulated world is the projection of this polygon onto the surface of the terrain along the Z-axis of the world coordinate system (§2.4).

The `breachedLaneID` field uniquely identifies the breached lane described by the Breached Lane PDU (§5.1.9). The id need only be unique within the exercise specified in the PDU header.

The `numberOfVertices` field specifies the number of vertices in the `vertices` field. It must be either three or four.

The `force` field identifies the force which created the breach lane (§5.1.7).

The `breachTime` field identifies the time the at which the breach lane was originally created. This time corresponds to the time the first Breached Lane PDU was issued for this `breachedLaneID`. The breach time is required to interpret the effect of multiple overlapping mine fields and breached lanes.

Marker PDU

The Marker PDU is used to introduce a static collection of objects (§5.1.10) into the simulated world. Its primary purpose is to represent the collection of marker flags that may be used to mark a mine field. A Marker PDU is issued whenever a collection of markers is created and once every 5 seconds (`vehicleAppearanceTime`) thereafter. A simulator may assume a collection of markers no longer exists if it has not received any Marker PDUs from that particular collection of markers for a period of 12 seconds (`vehicleDisappearanceTime`).

The Marker PDU is issued using the underlying datagram service. In addition to its PDU header, it includes the following fields:

```
type MarkerVariant sequence {
    guises              VehicleGuises,
    simulator           SimulationAddress,
    variantNumber       MarkerVariantNumber,
    numberOfMarkers     UnsignedInteger (8),
    force               ForceID,
    markers             array (numberOfMarkers) of MarkerDescriptor
}
```

The `guises` field contains two object type codes(§5.1.23). These object type codes specify the appearance of all the markers described by the Marker PDU. Which of the two type codes applies depends on the force (§5.1.7) to which the observer belongs. This dependency is explained in section 2.3.

The `simulator` field uniquely identifies the simulator which issued the Marker PDU (§5.1.13).

The `variantNumber` field logically groups and orders the markers listed in this Marker PDU with those in other Marker PDUs issued by the same simulator. The `variantNumber` field has the following form:

```
type MarkerVariantNumber sequence {
    number              UnsignedInteger (8),   -- the number of this one
    total               UnsignedInteger (8)    -- out of this many
}
```

The `number` field identifies this Marker PDU as the n^{th} member of a group of Marker PDUs. The `total` field identifies the total number of members of the group. Each

member of the group is uniquely identified by a `number` field in the range from 1 to the number of members of the group.

The `numberOfMarkers` field specifies the number of markers that follow in the `markers` field.

The `force` field identifies the force associated with the markers in this Marker PDU (§5.1.7).

Finally, each individual marker is described by a MarkerDescriptor data element in the `markers` field. The MarkerDescriptor data element has the following form:

```
type MarkerDescriptor sequence {
    location            WorldCoordinates,
    orientation         Angle,
    identifier          UnsignedInteger (16),
    unused (16)
}
```

The `location` field specifies the location of the marker in world coordinates (§5.1.29).

The `orientation` field specifies the orientation of the marker, measured counter-clockwise with respect to North. The angle is measured in units of BAMs (§5.1.1).

The `identifier` field uniquely identifies the marker within this simulator. When combined with the `simulator` field of the Marker PDU, it uniquely identifies the marker within the exercise specified in the PDU header. This combination of `simulator` and `identifier` is also unique over all vehicle identifiers (§5.1.24) and object identifiers (§5.1.9) in the same exercise.

7.3.4 Weapons fire

When a simulated vehicle fires its weapon, its simulator will usually issue two PDUs:

- The first, a Fire PDU, is issued when the shell or missile is fired.
- The second, an Impact PDU or Indirect Fire PDU, is issued later, when the projectile detonates.

The Fire PDU describes the type of projectile fired, the location of the muzzle or launcher from which it is fired, and the velocity of the projectile. This information is used by simulators receiving the PDU to display a muzzle flash near the appropriate point on the

firing vehicle. Also present in the Fire PDU to aid analysis of the exercise is information such as the rate of slew of the turret at the time of firing, the target range used for the fire control solution, and the kind of ammunition selected by the gunner.

In the case of direct fire where the projectile is either ballistic or it is guided in flight by the firing soldier or his simulator, the determination of what the projectile hits is done by the firing simulator. That simulator then issues an Impact PDU that describes the location of the projectile's impact and identifies any vehicle struck. Simulators receiving this PDU can display the impact and, if their vehicle is the one struck, assess any resulting damage.

In the case of indirect fire, the system simulating the howitzer or mortar issues an Indirect Fire PDU announcing the location of the projectile's detonation. In contrast with direct fire, no determination is made by the firing simulator as to which vehicles are hit by the indirect fire. Instead, each simulator computes its own vehicle's distance from the detonation and assesses any damage. Indirect Fire PDUs are also used to describe the detonations of bombs dropped by aircraft.

Although the Impact PDU and Indirect Fire PDU are optimized for describing the behavior of direct fire and indirect fire projectiles respectively, either PDU may be used with any given projectile type. The principle difference between the PDUs is in how a determination is made of what target vehicles are affected by a projectile. It may be desirable, for example, to issue an Impact PDU describing the detonation of a missile if it actually strikes a target vehicle, but to issue an Indirect Fire PDU instead if the detonation occurs some distance from any vehicle. In the former case, the target vehicle's simulator can assess its vehicle's damage most accurately, while in the latter case, the detonation can have an effect on several nearby vehicles.

Sometimes the protocol for a weapons engagement involves an Indirect Fire PDU but no preceding Fire PDU. This case arises, for example, when bombs are released by an imaginary aircraft that is not included in the simulation. When the bombs detonate, an Indirect Fire PDU is produced to describe them.

It is also possible for a round to be fired, yet never impact. This occurs, for example, when a round is fired outside the area of defined terrain. In such a case, an Impact PDU or Indirect Fire PDU is used to report the "nonimpact" of the round so that its disposition is never ambiguous.

Fire PDU

A Fire PDU describes the firing of a shell, a burst of machine gun fire, or a missile. It is issued by the firing simulator, and may be used by the simulators that receive it to display a muzzle flash at a location specified in the PDU.

A Fire PDU is issued using the underlying datagram service. In addition to its PDU header, it includes the following fields:

```
type FireVariant sequence {

    -- Common to all shell and missile firings:
    attackerID          VehicleID,
    eventID             EventID,
    burst               BurstDescriptor,
    target              TargetDescriptor,
    velocity            VelocityVector,
    muzzle              WorldCoordinates,
    projectileID        VehicleID,
                        unused (8),
    fireType            FireType,

    -- Depending on whether a shell or a missile is fired:
    specific            choice (fireType) of {

      -- If a shell is fired:
      when (fireTypeShell) shell sequence {
        range           Float (32),
        slewRate        Float (32),
        ammoSelected    ObjectType,
                        unused (32)
      },

      -- If a missile is fired:
      when (fireTypeMissile) missile sequence {
        tube            UnsignedInteger (8),
                        unused (56)
      }
    }
}
```

The `attackerID` field identifies the firing vehicle (§5.1.24).

The `eventID` field contains an event identifier generated by the firing vehicle's simulator (§5.1.5). It must be unique among all such identifiers generated for that vehicle since it joined the exercise. This same identifier is repeated in the subsequent Impact PDU or Indirect Fire PDU that reports the impact of the fired round.

The `burst` field identifies the kind of projectile and detonator fired, the number of rounds contained in a machine gun burst, and the rate of fire (§5.1.4).

The `target` field describes what the firing simulator knows about the target being fired upon (§5.1.16). It indicates whether the target is known, and, if so, whether it is a vehicle. If the target is known and it is a vehicle, the identifier of that vehicle is included in the field.

The `muzzle` field contains the location of the gun or missile launcher muzzle in world coordinates (§5.1.29). The velocity of the projectile leaving that muzzle is specified by the `velocity` field, in units of meters per second (§5.1.28). The vector is expressed in world coordinates (§2.4).

While modeling the flight of a projectile such as a missile, a simulator may produce Vehicle Appearance PDUs describing the appearance of the projectile. The `projectileID` field allows the firing simulator to specify a vehicle identifier it will be associating with the projectile in order to report its appearance as a vehicle (§5.1.24). If the firing vehicle's simulator will not be producing Vehicle Appearance PDUs describing the travelling projectile, it places zeros in the `projectileID` field.

The `fireType` field classifies the kind of ammunition fired as a shell or a missile. The values of this field are:

```
type FireType enum (8) {
    fireTypeShell (1),
    fireTypeMissile (2)
}
```

A machine gun burst is classified as `fireTypeShell`.

The format of the remaining portion of the PDU depends on whether the `fireType` field indicates that a shell or a missile is being fired. If a shell is being fired, the `range`, `slewRate`, and `ammoSelected` fields are present. The `range` field specifies the range (in meters) that the vehicle's fire control system has assumed in computing its

ballistic solution. The M1 tank simulator, for example, reports the range displayed in the gunner's sight. The `slewRate` field specifies the rate at which the vehicle's gun is slewing relative to the world coordinate system, in revolutions per second. The `ammoSelected` field specifies the type of ammunition that the vehicle's fire control system has assumed is being fired (§5.1.10). The M1 tank simulator, for example, reports the setting of the gunner's ammunition select switch.

Alternatively, if a missile is being fired, the `tube` field is present in the PDU. It specifies the tube from which the missile is being launched.

Impact PDU

An Impact PDU is issued by a simulator when the flight of a projectile it is simulating, ends. It may or may not describe an impact between the projectile and a particular, target vehicle. The PDU serves two purposes:

- It informs other simulators of the projectile's detonation so that they may produce the appropriate visual and aural effects.
- It may identify a specific target vehicle struck by the projectile so that the target vehicle's simulator can model the resulting damage.

There are two alternate ways in which the Impact PDU may be issued:

- If the projectile did not strike a target vehicle, or if it struck a target vehicle being simulated by the same simulator as that simulating the projectile, then the Impact PDU is issued using the underlying datagram service.
- Otherwise, (when the projectile struck a target vehicle being simulated by another simulator) the Impact PDU is issued using the underlying transaction service. The target vehicle's simulator is identified as the transaction's respondent. The response returned through via the transaction service contains no simulation PDU.

An Impact PDU includes the following fields in addition to its PDU header:

```
type ImpactVariant sequence {

    -- For any round fired:
    attackerID          VehicleID,
    eventID             EventID,
    burst               BurstDescriptor,
    projectileID        VehicleID,
    result              FireResult,
                        unused (8),

    -- For projectiles that impact somewhere:
    momentum            Float (32),          -- newton•seconds
    energy              Float (32),          -- joules
    directionality      Float (32),          -- steradians
    location            WorldCoordinates,
    range               Float (64),

    -- For shots that strike a particular vehicle:
    targetID            VehicleID,
    component           VehicleComponent,
    impact              VehicleCoordinates,
    trajectory          VehicleCoordinates
}
```

The `attackerID` field identifies the firing vehicle (§5.1.24).

The `eventID` field contains the same event identifier as that supplied by the firing simulator in the preceding Fire PDU (§5.1.5). This identifier, in conjunction with the firing vehicle's identifier, serves to associate corresponding pairs of Fire PDUs and Impact PDUs.

The `burst` field identifies the kind of projectile and detonator fired, the number of rounds contained in a machine gun burst, and the rate of fire (§5.1.4). It is identical to the `burst` field of the corresponding, preceding Fire PDU.

If the appearance of the fired projectile was described during its flight by a series of Vehicle Appearance PDUs, then the `projectileID` field contains the vehicle identifier used for the projectile (§5.1.24). Otherwise, the `projectileID` field contains zeros.

The `result` field indicates what has become of the fired round:

```
type FireResult enum (8) {
    nonImpact (1),          -- the projectile did not impact
    groundImpact (2),       -- the projectile struck terrain
    vehicleImpact (3),      -- the projectile struck a vehicle
    proximateImpact (4)     -- the projectile "struck" a vehicle
                            -- in proximity fused detonation

}
```

If the projectile was "lost" (e.g., it flew outside the area of defined terrain), then the `result` field will have the value `nonImpact`, and the remaining fields of the PDU are unused.

Otherwise, the `location` field specifies the point at which the projectile impacted either a vehicle or the terrain (§5.1.29), and the `range` field specifies the straight–line distance from muzzle to impact. The `momentum` and `energy` fields specify the impact momentum and explosive energy of the projectile, in newton·seconds and joules respectively. The directionality of the projectile's explosion is described by the `directionality` field, which specifies a solid angle in steradians. For a highly directional explosion, `directionality` will be small; for a spherically symmetrical explosion, `directionality` will be 4π.

Only if the projectile impacted a vehicle are the `targetID`, `component`, `impact`, and `trajectory` fields used. The `targetID` field identifies the vehicle struck (§5.1.24). The `component` field specifies which component of the target vehicle was struck. The `impact` field specifies the location at which the vehicle was struck, represented in its own vehicle coordinate system, as determined by the simulator producing the Impact PDU (§5.1.22). The `trajectory` field specifies the incident velocity of the projectile, also represented in the target vehicle's own coordinate system, in meters per second.

Indirect Fire PDU

The impacts of shells fired by howitzers or mortars, or the detonations of bombs dropped from aircraft, are described by Indirect Fire PDUs that are issued by the simulators modeling those artillery pieces or aircraft. A single Indirect Fire PDU can describe several bomb or shell detonations, specifying a location and time for each detonation. In contrast to the Impact PDU, this PDU does not identify the particular vehicle(s) affected by the detonations; every simulator computes its own vehicle's distance from the explosions in order to assess any damage.

Each detonation is described within the PDU by an Indirect Fire Detonation data element of the following form:

```
type IndirectFireDetonation sequence {
    location            WorldCoordinates,
    attackerID          VehicleID,
    eventID             EventID,
    delay               UnsignedInteger (16),
                        unused (40)
}
```

The `location` field of this object specifies the location of the detonation in world coordinates (§5.1.29). The use of the other two fields depends somewhat on the type of detonation being described. If the detonation is that of a shell fired by a mortar or howitzer, the `attackerID` field identifies the firing vehicle (§5.1.24), and the `eventID` field contains the event identifier used in the Fire PDU that previously reported the firing (§5.1.5).

Alternatively, if the detonation is that of a bomb, the `attackerID` field contains the vehicle identifier of the aircraft dropping the bomb. As there will be no preceding Fire PDU, the `eventID` field will be a new event identifier generated by the simulator modeling that aircraft.

In either case, the `vehicle` component of the `attackerID` vehicle identifier may instead contain zero if that firing vehicle or aircraft is purely notional (as is the case for close air support aircraft simulated by an MCC system).

A delay value is associated with each of the detonations described in the PDU. It specifies the amount of time, in milliseconds, by which that detonation follows the previous detonation. (The first delay value is the amount of time by which the first detonation follows issuance of the PDU.)

An Indirect Fire PDU is issued using the underlying datagram service.

Several detonations of the same kind may be described in a si[illegible]ect Fire PDU, which includes the following fields in addition to its PDU hea[illegible]

```
type IndirectFireVariant sequence {
    burst               BurstDescriptor,
                        unused (32),
    detonations         array (burst.quantity) of
                          IndirectFireDetonation
}
```

The `burst` field identifies the kind of projectile and fuze fired, and the number of detonations described by this PDU (§5.1.4). The `rate` component of the Burst Descriptor data element is not used and should contain the value `burstRateIrrelevant`.

A single Indirect Fire PDU may describe up to five detonations.

```
constant maxIndirectFireDetonations 5
```

7.3.5 Collisions

A Collision PDU is used to report collisions between vehicles. It serves two purposes: it ensures that when two vehicles collide, both are aware of the collision; and it allows the cause of vehicle damage to be identified.

There are two ways in which the Collision PDU is used:

- When any simulator becomes aware of a collision between its vehicle and a vehicle simulated elsewhere, it notifies the other simulator by using the underlying transaction service to convey a Collision PDU.
- When a simulator simulates a collision between two vehicles that it is both simulating, it reports the event by using the underlying datagram service to issue a Collision PDU.

Of course, only moving vehicles may cause collisions. The simulator of a moving vehicle must constantly check for collisions between its vehicle and either features of the terrain or other vehicles around it. However, the simulator of a vehicle that does not move (such as an MCC system simulating only static class vehicles) need not perform this processing. Instead it can learn of collisions involving its vehicle by listening for Collision PDUs.

A Collision PDU includes the following fields in addition to its PDU header:

```
type CollisionVariant sequence {
    vehicleID           VehicleID,
    eventID             EventID,
    targetID            VehicleID,
                        unused (16)
}
```

The `vehicleID` field identifies the vehicle whose simulator detected the collision (§5.1.24). That simulator will generate a unique event identifier and report it in the `eventID` field (§5.1.5). The `targetID` field identifies the other vehicle involved in the collision.

When a Collision PDU is issued using the supporting transaction service, it is issued as a transaction request; no simulation PDU is returned with the corresponding transaction response.

7.3.6 Transfer of munitions

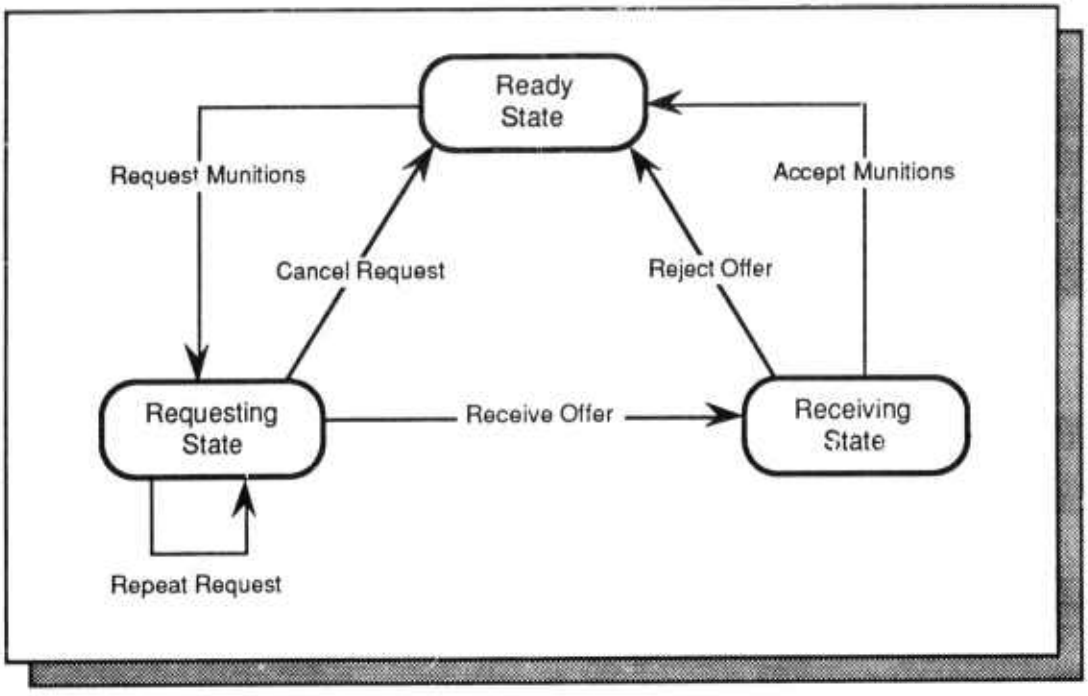
The simulation protocol provides a mechanism for transferring munitions, such as fuel or ammunition, between vehicles that are being modeled by separate simulators. For example, the protocol allows ammunition carriers and fuel tankers to resupply combat vehicles, and it allows combat vehicles to cross–level their ammunition loads among themselves.

The two vehicles participating in a transfer of munitions are referred to as the *supplier* and the *receiver*. The behavior of each is described by the state diagrams shown in figures 7–1 and 7–2. Briefly, the procedure is carried out as follows. At the beginning of the procedure, both the supplier and the receiver are in their respective ready states when the receiver notices a nearby vehicle capable of supplying it. The receiver transitions to requesting state upon requesting some munitions from the supplier, and it remains in that state while awaiting a reply to its request. When the reply arrives with an offer of munitions the receiver transitions to receiving state, and remains in that state for whatever time is required to load some portion of the offered munitions. After that time has elapsed, the receiver returns to ready state and sends an acknowledgement to the supplier for the portion of munitions taken. The supplier, meanwhile, waits in offering state from the time it offers the munitions until the time an acknowledgement is received from the receiver for some portion of those munitions.

We now describe this procedure in greater detail.

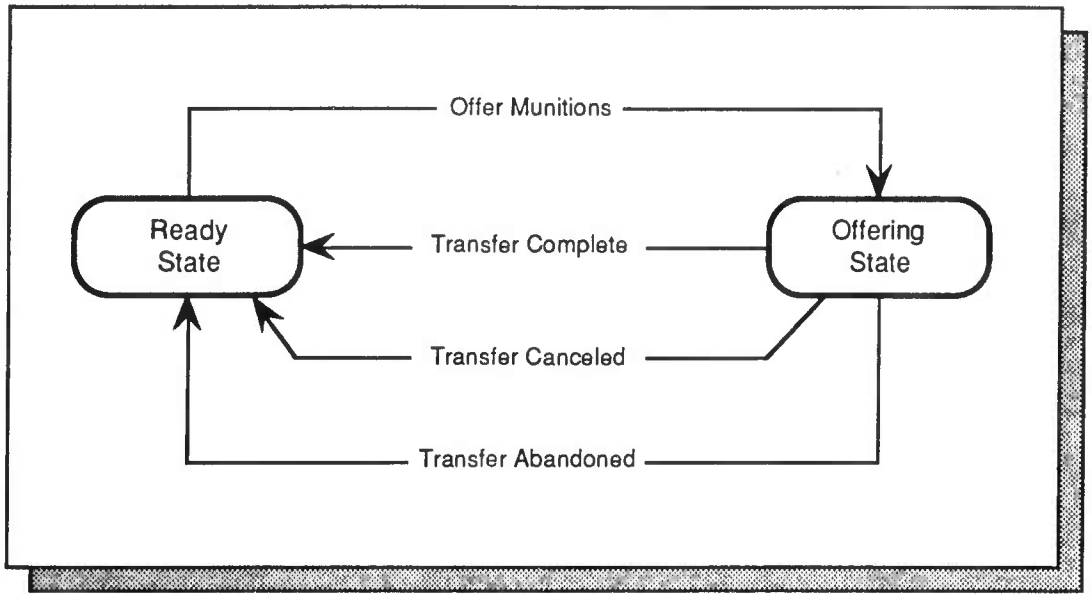
The procedure is initiated by a simulator (that of the receiver) issuing a Service Request PDU when it finds that the following conditions are all true:

- The receiver is in ready state.



Transition	Condition and Action
Request Munitions	When conditions for resupply are met, issue a Service Request PDU and set a timer to 5 seconds (`serviceRequestTime`).
Cancel Request	When conditions for resupply are no longer met, cancel the timer.
Repeat Request	When the timer expires, re-issue the Service Request PDU and reset the timer to 5 seconds (`serviceRequestTime`).
Receive Offer	When a Resupply Offer PDU is received, set the timer to the period required for receiving some increment of munitions.
Reject Offer	When conditions for resupply are no longer met, issue a Resupply Cancel PDU and cancel the timer.
Accept Munitions	When the timer expires, increment the count of munitions on board and issue a Resupply Received PDU.

Figure 7–1. Receiver behavior during a transfer of munitions



Transition	Condition and Action
Offer Munitions	When a Service Request PDU is received, issue a Resupply Offer PDU and set a timer to 1 minute (`resupplyTime`).
Transfer Complete	When a Resupply Received PDU is received, cancel the timer and decrement the count of munitions on board.
Transfer Canceled	When a Resupply Cancel PDU is received, cancel the timer.
Transfer Abandoned	When the timer expires.

Figure 7–2. Supplier behavior during a transfer of munitions.

- The receiver has the capacity for additional munitions, and it has identified a nearby vehicle capable of providing those munitions. For example, the receiver needs ammunition and a nearby vehicle has identified itself as an ammunition supplier by setting the `ammunitionSupplier` bit in the capabilities field of its Vehicle Appearance PDU (§5.1.19).
- The potential supplier is within an appropriate distance. For example, a fuel or ammunition truck must be within 30 meters of the M1 tank in order to supply it.
- Both receiver and supplier are stationary.
- Neither receiver nor supplier are destroyed.

- The transfer of munitions has been enabled by any necessary crew action appropriate to the receiver. For example, the transfer of ammunition to the M1 main battle tank is enabled when its crew set an ammunition resupply/distribution switch to a position labelled RECV.

Any of the above conditions pertaining to the supplier can be tested using information made available by the supplier's simulator in the form of Vehicle Appearance PDUs. When all conditions are satisfied, the receiver's simulator issues a Service Request PDU identifying the receiver, the potential supplier, and the kinds of munitions desired. Upon issuing the PDU, the receiver transitions to its requesting state.

A simulator that receives the Service Request PDU, and notices that its own vehicle is identified as the supplier, may respond by offering some portion of whatever munitions are currently loaded on that vehicle. In figure 8, this is shown as a transition from ready state to offering state. Meanwhile, the receiver's simulator re–issues its Service Request PDU every 5 seconds (`serviceRequestTime`) until such an offer is forthcoming. The offer takes the form of a Resupply Offer PDU issued by the supplier's simulator. It identifies the supplier, the receiver, and the quantities of various munitions offered. The munitions offered will be a subset of those possessed by the supplier, and a subset of those requested by the receiver.

Upon receiving the offer of munitions, the receiver changes from requesting state to receiving state. The receiver then has up to 1 minute (`resupplyTime`) to acknowledge the receipt of those munitions by returning to the supplier a Resupply Received PDU listing the exact munitions taken. The receiver need not accept all of the munitions offered, but instead can indicate in its receipt just how much it did accept. After delaying for up to 1 minute, the receiver issues its Resupply Received PDU and returns to ready state. When the supplier receives the Resupply Received PDU it also returns to ready state, and the procedure is complete.

The time required for the receiver to return the Resupply Received PDU, and the amount of munitions reported by that PDU as taken, determine the rate at which the supplier and receiver are able to transfer munitions. For example, an M1 tank obtaining 105 mm shells from an ammunition supply truck will acknowledge receipt of a single round after 40 seconds; this results in a simulated rate of resupply for the M1 tank of one round every 40 seconds.

Throughout the transfer process, both the receiver's and the supplier's simulators continue to monitor the conditions necessary for the transfer. If any of these conditions ceases to hold, either simulator can abort the transfer by issuing a Resupply Cancel PDU with the result that no munitions are transferred. Alternatively, the receiver can terminate the transfer early but accept some of the munitions offered by issuing a Resupply Received PDU for the partial load. Finally, if the supplier waits in offering state for a full minute (`resupplyTime`) but receives no Resupply Received PDU (perhaps because the receiver has withdrawn from the exercise), it should return to ready state and assume that no munitions were taken.[9]

The Service Request PDU, Resupply Offer PDU, and Resupply Received PDU are all of the same format. They identify the two participants in the transfer of munitions, and they list the munitions requested or transferred. These PDUs include the following fields in addition to their PDU headers:

```
type ResupplyVariant sequence {
    receiverID          VehicleID,
    supplierID          VehicleID,
    vehicleType         ObjectType,
    simulatorType       SimulatorType,
                        unused (40),
    numberMunitions     UnsignedInteger (8),
    munitions           array (numberMunitions) of MunitionQuantity
}
```

The `receiverID` and `supplierID` fields identify the receiving and supplying vehicles (§5.1.24). The vehicleType and simulatorType fields are used only for Service Request PDUs, where they identify the type of the receiving vehicle and the type of its simulator (§5.1.10 and §5.1.14). In Resupply Offer PDUs and Resupply Received PDUs, these fields contain zeros.

---

9 With the supply transfer protocol we have defined it is possible—although unlikely—that the supplier and receiver may disagree as to whether any supplies have been transferred. This situation will occur if a Resupply Received PDU becomes corrupted in transit, or if the supplier sends a Resupply Cancel PDU at the moment the receiver sends a Resupply Received PDU. These occurrences are too improbable to warrant the use of the more complex protocol that is necessary to prevent them.

The remaining portions of these PDUs list quantities of various types of munitions, representing each with a Munition Quantity data element (§5.1.8). The `numberMunitions` field reports the number of different types of munitions in this list, which may range from 0 to 28. Note that in a Service Request PDU soliciting munitions, there should always be at least one Munition Quantity data element present.

```
constant maxResupplyMunitions 28
```

The Resupply Cancel PDU includes the following fields in addition to its PDU header:

```
type ResupplyCancelVariant sequence {
    receiverID          VehicleID,
    supplierID          VehicleID,
                        unused (32)
}
```

The `receiverID` and `supplierID` fields identify the receiving and supplying vehicles (§5.1.24).

The Service Request PDU, Resupply Offer PDU, Resupply Received PDU, and Resupply Cancel PDU are all issued using the underlying datagram service.

7.3.7 Repairs

The simulation protocol allows one vehicle to be repaired by the crew of another. This feature is used to simulate maintenance teams that can reach disabled combat vehicles and carry out repairs to them under the direction of maintenance personnel. For a repair to be successfully performed, the maintenance team must remain with the disabled vehicle for a period of time determined by the nature of the repair. Once that period has elapsed, the repair is entirely accomplished. However, if the process is aborted before the necessary period has elapsed—perhaps due to the departure of the maintenance team—then no repair is performed and the combat vehicle is left in its initial state. There are no partial repairs.

We call the vehicle being repaired the *receiver*, and the vehicle whose crew is performing the repair, the *supplier*. The receiver's simulator is responsible for recognizing a nearby, potential supplier of repairs, and for identifying the times during which repairs can be

carried out. It does this by checking, every 5 seconds (`serviceRequestTime`), whether the following conditions are all true:

- The potential supplier is within an appropriate distance. For example, a maintenance team's vehicle must be within 30 meters of the M1 tank in order to repair it.
- Both receiver and supplier are stationary.
- Neither receiver nor supplier are destroyed.
- The performance of repairs has been enabled by any necessary crew action appropriate to the receiver. For example, the crew may be required to shut off their vehicle.

Any of the above conditions pertaining to the supplier can be tested using information made available by the supplier's simulator in the form of Vehicle Appearance PDUs.

If all conditions are satisfied, the receiver's simulator issues a Service Request PDU using the underlying datagram service. This PDU, whose format is defined in the previous section, carries the following information:

- The `receiverID` field identifies the vehicle soliciting service (the receiver).
- The `supplierID` field identifies the vehicle from which service is sought (the supplier).
- The `vehicleType` field identifies the type of vehicle soliciting service.
- The `simulatorType` field identifies the type of simulator modeling that vehicle.
- The `numberMunitions` field is 0, and the PDU contains no Munition Quantity data elements.

The receiver's simulator repeats its test of conditions every 5 seconds (`serviceRequestTime`), and each time it reissues the Service Request PDU if all conditions hold.

A simulator that receives the Service Request PDUs and notices that its own vehicle is identified as the supplier can allow the crew of its vehicle to perform a repair on the receiver. Any repair should be consistent with both the type of the receiver, and the type of the simulator modeling the receiver. As long as Service Request PDUs continue to be received, the repair process may be allowed to proceed. However, if Service Request PDUs cease to be received and are not seen for a period of 12 seconds (`serviceCancelTime`), the supplier's simulator must assume that the conditions listed above are no longer all true. It must therefore abort the repair process.

If the repair process successfully runs to completion, the supplier may then accomplish the repair by issuing a Repair Request PDU to notify the receiver of the repair. The receiver's simulator acknowledges receipt of the Repair Request PDU by returning a Repair Response PDU. (This acknowledgement simply indicates that the repair was performed, not that the repair was appropriate or that the disabled vehicle has become well because of the repair.) These two PDUs are exchanged using the underlying transaction service.

The Repair Request PDU includes the following fields in addition to its PDU header:

```
type RepairRequestVariant sequence {
    receiverID          VehicleID,
    supplierID          VehicleID,
    eventID             EventID,
    repair              RepairType
}
```

The `receiverID` field identifies the disabled vehicle, and the `supplierID` field identifies the maintenance team (§5.1.24). A unique event identifier is generated by the supplier's simulator and reported in the `eventID` field (§5.1.5).

The `repair` field identifies the type of repair performed (§5.1.12). The value of this field is interpreted according to the type of vehicle repaired, and the type of simulator modeling that vehicle. Repair codes appropriate to various types of vehicles and simulators are listed in appendix D.

The Repair Response PDU includes the following fields in addition to its PDU header:

```
type RepairResponseVariant sequence {
    receiverID          VehicleID,
    supplierID          VehicleID,
    result              RepairResult,
                        unused (24)
}

type RepairResult enum (8) {
    repairRequestAccepted,
    invalidRepairType
}
```

The contents of the `receiverID` and `supplierID` fields are identical to those of the corresponding Repair Request PDU. The `result` field indicates whether the repair request has been accepted, and, if not, why not.

8 DATA COLLECTION PROTOCOL

The data collection protocol is used to report, via the network, information about the simulated world. Whereas the simulation protocol conveys information of interest to simulators, the data collection protocol conveys additional information that is primarily of use to:

- Analysts who may be studying an exercise.
- Systems that must monitor the state of an exercise in order to restart it or resume it after some interruption.

8.1 Data collection protocol data units

The data collection protocol makes use of several kinds of protocol data units. All data collection PDUs have a length that is an integral multiple of 64 bits, and all begin with a common 64–bit header. Included in this header is a code indicating the kind of PDU present:

```
type DataCollectionPDUKind enum (8) {
    exerciseStatusPDUKind (1),          -- Exercise Status PDU
    simulationStatusPDUKind (2),        -- Simulation Status PDU
    vehicleStatusPDUKind (3),           -- Vehicle Status PDU
    statusQueryPDUKind (4),             -- Status Query PDU
    statusResponsePDUKind (5),          -- Status Response PDU
    statusChangePDUKind (6),            -- Status Change PDU
    laserRangePDUKind (7),              -- Laser Range PDU
    eventFlagPDUKind (8)                -- Event Flag PDU
}
```

PDUs containing an unknown `kind` field should be ignored. Kind values in the range of 129 to 255 are reserved for temporary or experimental use.

Following the PDU header is a portion whose format depends on the kind of PDU. The overall content of a PDU is:

```
type DataCollectionPDU sequence {
    version                 DataCollectionProtocolVersion,
    kind                    DataCollectionPDUKind,
    exercise                ExerciseID,
                            unused (40),
    variant                 choice (kind) of {

      when (exerciseStatusPDUKind)
        exerciseStatus      ExerciseStatusVariant,

      when (simulationStatusPDUKind)
        simulationStatus    SimulationStatusVariant,

      when (vehicleStatusPDUKind)
        vehicleStatus       VehicleStatusVariant,

      when (statusQueryPDUKind)
        statusQuery         StatusQueryVariant,

      when (statusResponsePDUKind)
        statusResponse      StatusResponseVariant,

      when (statusChangePDUKind)
        statusChange        StatusChangeVariant,

      when (laserRangePDUKind)
        laserRange          LaserRangeVariant,

      when (eventFlagPDUKind)
        eventFlag           EventFlagVariant
    }
}
```

The `version` field specifies the version of data collection protocol to which the PDU pertains. Use of this field allows new versions of the data collection protocol to be introduced without disruption to existing implementations. The data collection protocol described in this report has version number 3:

```
type DataCollectionProtocolVersion enum (8) {
    dcProtocolVersionAug89 (1),
    dcProtocolVersionJan90 (2),
    dcProtocolVersionJan90Corrected (3)
}
```

The `exerciseID` field identifies the exercise to which the PDU pertains (§5.1.6).

8.2 Use of association sublayer services

Data collection PDUs are conveyed among simulators using the services of the association sublayer defined in chapter 6. A single PDU is issued through a single invocation of either the A–Datagram.req service primitive or the A–Transact.req service primitive. The discussion of protocol procedures, below, specifies which of these two service primitives is used in each case.

To distinguish the data collection protocol from other protocols using the association sublayer, the data collection protocol is assigned a unique association sublayer user protocol number. This number is 2:

```
constant dataCollectionProtocolNumber 2
```

Most data collection protocol interactions among simulation entities take place within the context of a particular simulation exercise. Associated with an exercise is an exercise identifier, which distinguishes it from other, concurrent exercises. With but one exception noted below, the interactions associated with a particular exercise are carried by the association service using a multicast group number that is identical to the exercise's identifier. This allows simulation entities to receive information only about the exercises of interest to them by subscribing only to selected multicast groups.

The one exception to the rule stated above on the use of exercise identifiers as multicast group numbers occurs with the protocol interaction used to query simulation entities about the exercises they are participating in. That protocol interaction may be performed using the multicast group number 0 (a group that includes all simulation entities) to ensure that a simulation entity may be queried regardless of which multicast groups it has already subscribed to.

The transaction service of the association sublayer allows a respondent to specify whether a particular response should be cached for retransmission. This service element is chosen using the cache–response parameter of the A–Transact.rsp service primitive. Use of this feature for data collection protocol interactions is optional in each case, unless specified otherwise in the following description of that protocol interaction.

8.3 Protocol procedures

The data collection protocol is logically divided into two distinct protocol procedures, each of which provides a related set of functions. One protocol procedure provides for reporting information about the internal state of simulators. Another provides for reporting interesting events. The two sets of protocol functions are described in separate sections, below.

All protocol procedures involve the exchange of PDUs among simulation entities using services of the association sublayer. For brevity, we sometimes use the term *simulator* in this section when referring to simulation entities. When describing use of the association sublayer, however, we generally revert to the more correct term, *simulation entity*. Moreover, we usually shorten "the transaction service of the association sublayer" to "the underlying transaction service".

8.3.1 Status reports

One purpose of the data collection protocol is to allow information to be obtained about the conditions of an exercise and the state of simulators to supplement that which is already available via the simulation protocol. The data collection protocol requires simulators to periodically report internal state information, as well as to report this information in response to certain prompting PDUs.

In the following sections, we first describe the PDUs that are used by simulators to report information. We then describe the conditions under which these PDUs are issued.

Exercise Status PDU

An Exercise Status PDU describes the conditions being simulated in an exercise. In addition to its PDU header, the PDU includes the following fields:

```
type ExerciseStatusVariant sequence {

    -- Times:
    realTime            Time,
    simulatedTime       Time,

    -- The terrain database chosen for the exercise:
    terrain             TerrainDatabaseID,

    -- The battle scheme chosen for the exercise:
    battleScheme        BattleScheme,

    -- Weather conditions:
    simulatedSkyColor   SkyColor,
                        unused (16),
    VLVisibility        Float (32), -- visibility in visible light in meters

}
```

The PDU's header identifies the exercise described by the PDU (§5.1.6).

The `realTime` and `simulatedTime` fields report the real (actual) time, and the simulated time, as of the moment the PDU is issued (§5.1.18). The terrain upon which the exercise is taking place is identified by the `terrain` field (§5.1.17). How force identifiers and guises are being used is identified by the `battleScheme` field (§5.1.2). The weather conditions are described by the `simulatedSkyColor` and `VLVisibility` fields. These two fields are used in the same manner as in the Activate Request PDU (§7.3.1).

Simulation Status PDU

A Simulation Status PDU describes the role a particular simulator is performing in an exercise, and the parameters according to which it is conducting its simulation.

A Simulation Status PDU includes the following fields in addition to its PDU header:

```
type SimulationStatusVariant sequence {
    simulator           SimulatorType,
    numberUnits         UnsignedInteger (8),
                        unused (8),

    -- Times:
    exerciseEntryTime   Time,
    realTime            Time,
    simulatedTime       Time,

    -- The terrain database used by the simulator:
    terrain             TerrainDatabaseID,

    -- The organizational units simulated:
    units               array (numberUnits) of OrganizationalUnit,

    -- Simulator-specific information:
    specific            choice (simulator) of {

      when (simulator_SIMNET_MCC)
        simnetMCC       SIMNET_MCC_Status,

    }
}
```

The `simulator` field describes the type of simulator (§5.1.14), and the PDU's header identifies the exercise in which it is participating (§5.1.6). Three fields are included for reporting times (§5.1.18):

`exerciseEntryTime`	is the real time at which the simulator became a participant in the exercise.
`realTime`	is the real time at which the PDU is issued.
`simulatedTime`	is the simulated time at which the PDU is issued.

Any of these three fields may contain 0 if its correct value cannot be determined by the simulator.

The `terrain` field identifies the terrain database being used by the simulator, by name and version number (§5.1.17). In this use of the Terrain Database ID data element, the version number reported in the `terrainVersion` field of that data element may not be the value `latestTerrainVersion` (defined as 0).

The `units` field is an array that lists the organizational units being modeled by the simulator. Each array entry is an Organizational Unit data element that identifies one unit and specifies the force to which the unit is assigned (§5.1.11). If individual components of a unit are assigned to different forces, the `force` field of its Organizational Unit data element contains the value `forceIDIrrelevant` (defined as 0). The number of units described is specified by the `numberUnits` field.

The last portion of the PDU may contain information specific to the type of simulator reporting. At present one variant is defined, for use by the SIMNET MCC system:

```
type SIMNET_MCC_Status sequence {

    -- Whether various optional elements are being simulated:
    optionTOC           Boolean,            -- Tactical Operations Center
    optionAdminLog      Boolean,            -- Admin/Log Center
    optionBnHQTanks     Boolean,            -- Bn HQ tank section
    optionScoutPlt      Boolean,            -- scout platoon
    optionFSE           Boolean,            -- fire support element
    optionALO           Boolean,            -- air liaison officer
    optionCSS           Boolean,            -- combat service support
    optionCE            Boolean,            -- combat engineering
                        unused (56)
}
```

The Boolean fields labelled `optionTOC` through `optionCE` specify whether various optional elements that the MCC system can simulate are actually enabled for use in the exercise. A bit is 1 if its corresponding element is enabled, and 0 otherwise.

Vehicle Status PDU

The status of an individual vehicle is reported using a Vehicle Status PDU. In addition to its PDU header, the PDU includes the following fields:

```
type VehicleStatusVariant sequence {
    vehicleID           VehicleID,
                        unused (16),
    unit                OrganizationalUnit,
    status              VehicleStatus,
                        unused (32)
}
```

The PDU's header identifies the exercise in which the vehicle is participating (§5.1.6). Its vehicle identifier is contained in the `vehicleID` field, and the `unit` field specifies the force and organizational unit to which the vehicle has been assigned (§5.1.11). The operational status of the vehicle, the health of each of its subsystems, and the quantities of various supplies it carries are all represented in the `status` field (§5.1.26).

When status is reported

Certain status PDUs (Exercise Status, Simulation Status , and Vehicle Status PDUs) contain information that is essential for restarting a simulator after an equipment failure, or for restoring an exercise to earlier conditions. These PDUs are issued at regular intervals chosen according to the volatility and quantity of the information represented. They are also issued in response to queries, as described later in this section. All simulators respond to the appropriate queries, but some also issue status PDUs at regular intervals according to the following conventions:

- A crewed vehicle simulator that models its vehicle's individual munitions stores and subsystems issues a Vehicle Status PDU every 30 seconds (`vehicleStatusTime`) to report the state of that vehicle.
- A simulator that does not model the internal stores and subsystems of a vehicle need not periodically issue Vehicle Status PDUs for that vehicle.
- A simulator that models an entire organizational unit issues a Simulation Status PDU every 5 minutes (`simulationStatusTime`) to identify its role.
- A simulator that does not model an entire organizational unit need not periodically issue Simulation Status PDUs.
- Exercise Status PDUs are not issued periodically by any simulator (they are issued only in response to queries).

A status PDU is also issued by a simulator in response to an appropriate query in the form of a Status Query PDU. A Status Query PDU includes the following fields in addition to its PDU header:

```
type StatusQueryVariant sequence {

    -- Specifying the type of response sought:
    responseKind        DataCollectionPDUKind,

    -- Specifying the desired respondents:
    unitRelation        UnitRelation,
    simulatorType       SimulatorType,
    vehicleID           VehicleID,
                        unused (16),
    unit                OrganizationalUnit
}

type UnitRelation enum (8) {
    unitRelationIrrelevant,
    unitSpecified,
    unitIncluded,
    unitIncluding
}
```

The fields of the Status Query PDU allow the querying simulator to specify the type of status information sought, and to identify, by various means, the simulators that should provide it. By using appropriate values in these fields, a simulation entity can issue a Status Query PDU using the underlying datagram service and discover what other simulation entities are present by the status PDUs they return. Alternatively, a simulation entity can send a Status Query PDU to a particular other simulation entity using the underlying transaction service in order to solicit a status PDU only from that simulator.

The `responseKind` field indicates the type of information sought by specifying a kind of data collection PDU. It contains one of the values `exerciseStatusPDUKind`, `simulationStatusPDUKind`, or `vehicleStatusPDUKind`.

Other fields of the Status Query PDU allow any of five conditions to be specified. Each condition may be specified or not, at the discretion of the simulator issuing the query. A simulator responds only if it meets all of the conditions specified in the query.

1. The `exerciseID` field of the Status Query PDU's header provides one mechanism by which respondents are selected (§5.1.6). If the `exerciseID` field has the value `exerciseIDIrrelevant` (defined as 0), and the PDU is issued to the association sublayer using multicast group 0, then a simulator involved in any exercise may respond. Otherwise, the `exerciseID` field specifies a particular exercise, the PDU is issued using the corresponding multicast group, and only simulators participating in that exercise will respond.

2. The `vehicleID` field allows the query to be directed to the simulator modeling a particular vehicle (§5.1.24). If the `vehicleID` field contains zeros, then a simulator may respond regardless of which vehicles it models. Otherwise, a simulator may respond only if it models the specified vehicle.

3. The `simulatorType` field allows the query to be directed to a particular type of simulator (§5.1.14). If the `simulatorType` field has the value `simulatorUnknown` (defined as 0), then any type of simulator may respond. Otherwise, a simulator may respond only if it is of the specified type.

4. The `force` component of the `unit` field (§5.1.11) allows the query to be directed to simulators of vehicles assigned to a particular force. If the `force` component has the value `forceIDIrrelevant`, then a simulator may respond regardless of what forces its vehicles are assigned to. Otherwise, a simulator may respond only if it simulates a vehicle assigned to the specified force.

5. The `unitRelation` and `unit` fields allow the query to select respondents based on the organizational units they simulate. The `unitRelation` field specifies one of these four cases:

`unitRelationIrrelevant`	a simulator may respond regardless of what unit it simulates.
`uritSpecified`	a simulator may respond if it simulates the specified unit.
`unitIncluded`	a simulator may respond if it simulates a unit that is included within that specified by the `unit` field.
`unitIncluding`	a simulator may respond if it simulates a unit that includes that specified by the `unit` field.

If a query specifies that a response is to be in the form of a Vehicle Status PDU, and the conditions specified in the query include those that pertain to individual vehicles (selecting on the basis of vehicle identifier, force, or organizational unit), then a responding simulator issues Vehicle Status PDUs only for those vehicles that meet the query's conditions.

If a Status Query PDU is issued using the association sublayer's datagram service, then any response is also issued using the datagram service. A response is issued with the same multicast group number as the query. The simulator issuing the query waits up to 5 seconds (`statusQueryTime`) for any responses to be received.

If a Status Query PDU is issued using the association sublayer's transaction service, then the only simulator that may return a response is that selected as the transaction's respondent. In this case, if the conditions specified in the Status Query PDU are met, then the respondent returns the appropriate status PDU. In doing so, the responding simulation entity does not permit its association entity to cache its response, but rather, it generates a new, fully current status PDU whenever its response must be reissued. If the conditions specified in the Status Query PDU are not met, the respondent returns a Status Response PDU. In addition to its PDU header, the Status Response PDU includes the following field:

```
type StatusResponseVariant sequence {
    result              StatusResult,
                        unused (56)
}

type StatusResult enum (8) {
    invalidQueryParameter (1),
    queryConditionFailed (2)
}
```

The `exerciseID` field of the PDU's header contains the exercise identifier that was included in the corresponding Status Query PDU. The `result` field of this PDU contains the value `invalidQueryParameter` if the Status Query PDU contained an invalid parameter, and the value `queryConditionFailed` otherwise.

8.3.2 Event reports

The data collection protocol includes provisions for reporting various types of events that may be of interest to those studying an exercise. Changes in a vehicle's operational status are reported by the vehicle's simulator using Status Change PDUs. These PDUs allow an analyst to follow the fortunes of an individual vehicle, and attribute changes in its operational status to specific causes. The use of a vehicle's laser range finder is reported by a Laser Range PDU whose contents may allow an analyst to determine what target was examined and what result was obtained. Another PDU, the Event Flag PDU, is available to the analyst for annotating a recording of an exercise (such as that produced by the Data Logger) to flag any event that may be of interest.

Reporting a change of status

At the time that the operational status of a vehicle or any of its subsystems changes, the vehicle's simulator issues a Status Change PDU describing what has changed, and why. This PDU is issued when any of the following events occurs:

- A vehicle's subsystem fails for any reason, including wear–and–tear, a collision, crew error, interaction with the terrain, or combat damage.
- A vehicle's subsystem returns to operation for any reason, including crew repair or repair by a maintenance team.
- A vehicle is destroyed by any cause.
- A vehicle is reconstituted by any cause.

In addition to its PDU header, the Status Change PDU includes the following fields:

```
type StatusChangeVariant sequence (
    vehicleID           VehicleID,
                        unused (8),
    effect              StatusChangeEffect,
    cause               choice (effect) of (

      when (effectVehicleDestroyed) destroyed sequence (
        kind            DamageCause,
                        unused (24)
      },

      when (effectVehicleReincarnated) reincarnated sequence (
        kind            RepairCause,
                        unused (24)
      },

      when (effectSubsystemsDamaged) damaged sequence (
        kind            DamageCause,
                        unused (24)
      },

      when (effectSubsystemsRepaired) repaired sequence (
        kind            RepairCause,
                        unused (24)
      }
    },
    eventID             EventID,
    agentID             VehicleID,
    subsystems          VehicleSubsystems
}

type StatusChangeEffect enum (8) {
    effectVehicleDestroyed (1),
    effectVehicleReincarnated (2),
    effectSubsystemsDamaged (3),
    effectSubsystemsRepaired (4)
}
```

```
type DamageCause enum (8) {
    damageCauseOther (0),                    -- none of those listed below
    damageCauseBreakdown (1),
    damageCauseCollision (2),
    damageCauseCrewError (3),
    damageCauseDirectFire (4),
    damageCauseIndirectFire (5),
    damageCauseIntervention (6),
    damageCauseTerrain (7),
    damageCauseBattlemaster (8)
}

type RepairCause enum (8) {
    repairCauseOther (0),                    -- none of those listed below
    repairCauseCrew (1),
    repairCauseIntervention (2),
    repairCauseMaintTeam (3),
    repairCauseBattlemaster (4)
}
```

The `vehicleID` field identifies the affected vehicle (§5.1.24), and the `effect` field describes how it was affected:

`effectVehicleDestroyed`	the vehicle was destroyed.
`effectVehicleReincarnated`	the vehicle was restored to complete operation.
`effectSubsystemsDamaged`	some vehicle subsystems have been damaged.
`effectSubsystemsRepaired`	some vehicle subsystems have been repaired.

In the case of vehicle subsystems becoming damaged or repaired, the particular subsystems involved are identified by the `subsystems` field (§5.1.27). That field contains a single Boolean data element for each of the vehicle subsystems whose operational status is being modeled by the simulator. A Boolean is true if the corresponding subsystem has been affected (damaged or repaired), and false otherwise. Also present in the field are Boolean data elements that summarize groups of subsystems. If a vehicle loses or gains its firepower ability, for example, the `firepowerSummary` component of the Vehicle Subsystems data element will contain the value true (defined as 1).

The `cause` field describes what caused the change of operational status reported by the PDU. If the effect is one of `effectVehicleDestroyed` or

`effectSubsystemsDamaged`, then the `cause` field contains one of the following values:

`damageCauseOther`	none of the causes listed below.
`damageCauseBreakdown`	the breakdown was due to random failure.
`damageCauseCollision`	the vehicle collided with another.
`damageCauseCrewError`	the damage was caused by crew error.
`damageCauseDirectFire`	the vehicle was struck by direct fire.
`damageCauseIndirectFire`	the vehicle was damaged by indirect fire.
`damageCauseIntervention`	the change was due to intervention by a technician.
`damageCauseTerrain`	the vehicle collided with a terrain feature, such as a building.
`damageCauseBattlemaster`	the change was due to intervention by the Battlemaster (exercise controller)

If the effect is one of `effectVehicleReincarnated` or `effectSubsystemsRepaired`, then the `cause` field contains one of the following values:

`repairCauseOther`	none of the causes listed below.
`repairCauseCrew`	the damage was repaired by the vehicle's own crew.
`repairCauseIntervention`	the change was due to intervention by a technician.
`repairCauseMaintTeam`	the damage was repaired by a simulated maintenance team.
`repairCauseBattlemaster`	the change was due to intervention by the Battlemaster (exercise controller)

Some changes in the operational status of a vehicle may be attributed to an event reported by an earlier simulation protocol PDU. If the vehicle is damaged in a collision with another, there will have been a Collision PDU issued by either or both vehicles' simulators, reporting that collision. If a vehicle is damaged by a missile or shell, there will have been a Vehicle Impact PDU or Indirect Fire PDU produced, describing the impact or explosion. A simulated repair to a vehicle that succeeds in fixing any of its subsystems will have been described by a Repair PDU. In all of these cases, the

preceding PDU reporting the causal event contains a unique combination of vehicle and event identifiers, distinguishing that event from all others. The Status Change PDU specifies that event by repeating the vehicle and event identifiers in its `agentID` and `eventID` fields (§5.1.24 and §5.1.5).

In other cases, where no preceding PDU can be said to represent the cause of the status change, the `agentID` field contains zeros and `eventID` field contains the value `eventIDIrrelevant`.

Laser range finding

The Laser Range PDU reports a vehicle's use of its laser range finder. The M1 tank simulator, for example, issues an instance of this PDU whenever one of the simulator's laser range finder buttons is pressed by a crew member. In addition to its PDU header, the PDU includes the following fields:

```
type LaserRangeVariant sequence {
    vehicleID           VehicleID,
    result              LaserRangeResult,
    returnSwitch        ReturnSwitch,
    target              TargetDescriptor,
    muzzle              WorldCoordinates,
    location            WorldCoordinates,
    whichLaserRange     LaserRangeFinder (8),
                        unused (56)
)

type LaserRangeResult enum (8) {
    laserRangeMalfunction (1),
    laserRangeMultipleReturns (2),
    laserRangeNoReturn (3),
    laserRangeSingleReturn (4)
)
```

The `vehicleID` field identifies the vehicle using its range finder (§5.1.24), and the `result` field describes the effect of using it. The `result` field's interpretation is:

`laserRangeMalfunction`	an attempt was made to obtain a range, but the rangefinder malfunctioned.
`laserRangeMultipleReturns`	multiple returns were received by the laser rangefinder; one is reported in this PDU.

`laserRangeNoReturn`	no acceptable returns were received by the laser rangefinder.
`laserRangeSingleReturn`	a single return was received by the laser rangefinder; it is reported in this PDU.

If the vehicle has a switch for selecting whether the range finder should report the first or the last return it receives, the position of this switch is indicated in the `returnSwitch` field. It contains one of these values:

```
type ReturnSwitch enum (8) {
    noReturnSwitch,                    -- no "first/last return" switch
    firstReturn,                       -- switch set to "first return"
    lastReturn                         -- switch set to "last return"
}
```

The location of the laser's "muzzle", in world coordinates, is described by the `muzzle` field (§5.1.29).

The `whichLaserRange` field identifies which of the vehicle's laser range finders was employed. The laser range finders are enumerated as follows:

```
type LaserRangeFinder enum (8) {
    gunnerLaser,        -- gunner's laser
    citvLaser           -- Commander's Independent Thermal Viewer laser
}
```

If the range finder was able to deduce a range (in which case the PDU's result field contains either `laserRangeSingleReturn` or `laserRangeMultipleReturns`) then two other fields of the PDU will also contain information. The `target` field will specify whether the lased target is known, and if so, what it is (§5.1.16). The `location` field will specify the point, in world coordinates, whose range was reported by the range finder (§5.1.29).

Flagging events of interest

The Event Flag PDU may be used to insert an annotation at a particular point in a recording of an exercise being made by a Data Logger. Typically, the PDU marks an event that is of interest to an analyst studying an exercise, such as the transmission of an

order by voice radio. Upon hearing the transmission, the analyst may cause an appropriate Event Flag PDU to be issued. The PDU will be recorded by the Data Logger in sequence with other PDUs related to the exercise, thereby identifying that event in the context of the overall exercise.

An Event Flag PDU includes the following fields, in addition to its PDU header:

```
type EventFlagVariant sequence (
    number              Integer (32),
    sequenceNumber      UnsignedInteger (32),
    vehicleID           VehicleID,
    textLength          UnsignedInteger (16),
    text                array (textLength) of Character (8)
)
```

The `number` field contains an integer, assigned by the analyst, identifying the type of event being flagged. The `sequenceNumber` field contains consecutive integers in consecutive Event Flag PDUs produced by the same simulation entity. It permits the detection of cases where Event Flag PDUs may have been lost. The `vehicleID` field may be used to specify a vehicle (§5.1.24); if not used for that purpose, it contains zeros.

The `text` field may contain any ASCII character string, up to a maximum of 228 characters. If necessary, the `text` field is followed by up to 56 unused bits so that the overall size of the PDU is a multiple of 64 bits.

9 REFERENCES

[1] James Chung, Alan Dickens, Brian O'Toole, and Carol Chiang. *SIMNET M1 Abrams Main Battle Tank Simulation: Software Description and Documentation (Revision 1).* BBN Report Number 6323. BBN Systems and Technologies Corp. Cambridge, Mass., August 1988.

[2] James Chung, Alan Dickens, Carol Chiang, Brian O'Toole, Warren Katz, and Bryant Collard. *SIMNET M2/3 Bradley Fighting Vehicle Simulation: Software Description and Documentation.* BBN Report Number 6892. BBN Systems and Technologies Corp. Cambridge, Mass., August 1988.

[3] Arthur Pope. *The SIMNET Network and Protocol.* BBN Report Number 6369. BBN Laboratories, Inc. Cambridge, Mass., February 1987.

[4] Arthur Pope. *The SIMNET Network and Protocols.* BBN Report Number 6787. BBN Laboratories, Inc. Cambridge, Mass., May 1988.

[5] A. Ceranowicz, S. Downes–Martin and M. Saffi. *SIMNET Semi–Automated Forces Version 3.0: A Functional Description (Revised).* BBN Report Number 6939. BBN Systems and Technologies Corp. Cambridge, Mass., March 1989.

[6] Arthur Pope, Tim Langevin, Linda Lovero and Andrew Tosswill. *The SIMNET Management, Command and Control System.* BBN Report Number 6473 (Revised). BBN Systems and Technologies Corp. Cambridge, Mass., July 1988.

[7] C. Topolcic, Ed, *Experimental Internet Stream Protocol, Version 2 (ST-II)*, RFC 1190. CIP Working Group, October 1990.

[8] International Standards Organization. *Information processing systems — Open Systems Interconnection — Basic Reference Model.* ISO 7498-1984.

[9] International Standards Organization. *Technical Report 8509 — Information Processing Systems Open Systems Interconnection — Service Conventions.* 1985.

[10] The Institute of Electrical and Electronics Engineers, Inc. *Standard for Binary Floating–Point Arithmetic.* ANSI/IEEE Standard 754–1985. The Institute of Electrical and Electronics Engineers, Inc. New York, New York, 1985.

[11] *The Ethernet: A Local Area Network: Data Link Layer and Physical Layer Specifications*. Digital Equipment Corporation, Intel Corporation, and Xerox Corporation; Version 2.0; November 1982.

[12] The Institute of Electrical and Electronics Engineers, Inc. *IEEE Standards for Local Area Networks: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*. ANSI/IEEE Std 802.3-1985. The Institute of Electrical and Electronics Engineers, Inc. New York, New York, 1985.

[13] Dan Friedman and Varda Haimo. *SIMNET Network Performance*. BBN Report Number 6711. BBN Communications Corporation. Cambridge, Mass., 1988.

[14] The Institute of Electrical and Electronics Engineers, Inc. *IEEE Standards for Local Area Networks: Logical Link Control* ANSI/IEEE Std 802.2-1985. The Institute of Electrical and Electronics Engineers, Inc. New York, New York, 1984

[15] The Institute of Electrical and Electronics Engineers, Inc. *Project 802: Local and Metropolitan Area Network Standard - Overview and Architecture* IEEE P802.1A/D10. The Institute of Electrical and Electronics Engineers, Inc. New York, New York, 1990.

[16] J.K. Reynolds and J. B. Postel. *Assigned numbers*, Request For Comments 1060, Internet Activities Board. March, 1990.

[17] Arthur Pope. *The SIMNET Network and Protocols*. BBN Report Number 7102. BBN Laboratories, Inc. Cambridge, Mass., July 1989.

APPENDIX A: DATA REPRESENTATION NOTATION

A formal notation is used in this report to define the format of communicated data. This appendix describes that notation, which we call Data Representation Notation (DRN).

In the following description of DRN, fragments of the notation are displayed in a "typewriter" font like this: `sequence`. Within a fragment of notation, a name displayed in italics (e.g., *`fieldName`*) is a placeholder for which another fragment of notation is to be substituted.

A.1 Overview

A *DRN specification* consists of a series of *type definitions* and *constant definitions*. A type definition describes how a unit of data is represented as a sequence of bits, and it associates a name with that representation. A constant definition associates a name with a particular value. The type definitions and constant definitions published in this report constitute a single DRN specification.

The notation employs these kinds of lexical tokens: reserved words, names, numbers, and delimiters.

The reserved words are: `array`, `choice`, `constant`, `enum`, `of`, `sequence`, `type`, `unused`, `when`, `Character`, `Float`, `Integer`, `UnsignedInteger`.

A name consists of a letter followed by zero or more letters, digits, and "_" characters. Both uppercase and lowercase letters may be used. A name must differ from any reserved word.

A number is a series of one or more digits. It is interpreted as a decimal value.

The delimiters are: "(", ")", "--", ",", ".".

The "--" delimiter is used to mark the presence of a comment, which lies between the "--" delimiter and the end of the line. Apart from their role in terminating comments, ends of lines are of no significance. Blanks (spaces) may appear anywhere, and are only required in certain places to delimit reserved words, names, and numbers. The other delimiters will be described in the context of their usage.

The "@" character is reserved for the purpose of annotating DRN specifications with control information intended to direct automated software tools. A line beginning with "@" followed by a name is interpreted as such a directive. This report does not define specific directives of this form.

A.2 Constant definition

A constant definition associates a names with a numeric value. It is of the form:

```
constant name number
```

The name must be one not defined elsewhere in the DRN specification by either a constant definition or a type definition. By convention, constant names begin with lower–case letters.

A.3 Type definition

A type describes a data representation as a way of encoding information in a series of bits. A type definition, which associates a name with a particular type, is of the form:

```
type name typeSpec
```

The *name* must be one not defined elsewhere in the DRN specification by either a constant definition or a type definition. By convention, type names begin with upper–case letters.

The *typeSpec* specifies a particular type. There are three ways in which a type may be specified: through the use of a *primitive type*, through the use of a *type constructor*, or by reference to a name defined elsewhere in a type definition. Primitive types and type constructors are described in the following sections.

A.4 Primitive types

Primitive types are the basic data elements out of which other, more elaborate data elements may be constructed. There are five kinds of primitive type. Four of these are specified by a particular reserved word, followed by a number in parentheses:

```
Character ( bits )
Float ( bits )
Integer ( bits )
UnsignedInteger ( bits )
```

In each case, the number, *bits*, specifies the quantity of consecutive bits to be used to represent the data element. The octets of data elements larger than 8 bits are represented in order, with the most–significant octet first and the least–significant octet last.

The Character primitive type represents a printable character. Its size is always specified as 8 bits. The character is represented using ASCII encoding, with the high–order bit always 0.

The Float primitive type represents a floating point number encoded in ANSI/IEEE standard format [10]. Its size is always specified as either 32 bits or 64 bits.

The Integer primitive type represents a signed integer encoded in twos–complement binary format. It never exceeds 32 bits.

The UnsignedInteger primitive type represents an unsigned binary integer. It never exceeds 32 bits.

The fifth kind of primitive type, called an enumerated type, specifies a list of possible values and the number of bits used to represent them. It is written:

```
enum ( bits ) { valueList }
```

As before, *bits* specifies the quantity of consecutive bits to be used to represent the data element. This number never exceeds 32. The data element represents a value as a unsigned binary integer. The possible values of the data element are enumerated by the *valueList*, which is a series of comma–separated items. Each item may be either of the form:

```
valueName ( number )
```

or just:

```
valueName
```

Each item in the series associates a name, *valueName*, with a numeric value. By convention, value names begin with lower–case letters. These names must be unique,

and not defined elsewhere in the DRN specification by either a constant definition or a type definition. If an item specifies both a name and a number, then the name is associated with the specified number. Otherwise, the name is associated with a number one larger than that associated with the previous name in the series. If no number is specified for the first name in the series, then that name is associated with the value 0.

A.5 Sequence type constructor

A *sequence* is one kind of type constructor. It specifies an ordered collection of various types of data elements called *fields*, and it is written:

```
sequence { fieldList }
```

where *fieldList* is a series of comma–separated items. Each item may be either of the form:

```
fieldName typeSpec
```

or of the form:

```
unused ( bits )
```

An item of the first form specifies that a data element of a particular type is present, and it associates a name with that data element. The *typeSpec* may be a primitive type, a type constructor, or a name defined by a type definition. The name must be unique among those included in the *fieldList*. By convention, field names begin with lower–case letters.

The second form of field list item is used to specify that some bits of the sequence remain unused. These bits appear in the representation, but no data is encoded by them.

The successive fields of a structure occupy successive bits in the representation. The bits of a particular field will precede all other bits of fields defined later in the same structure. Within an octet, bits are allocated to fields in order from most– to least–significant bit.[10]

---

[10] Do not confuse the order in which bits are allocated to the fields of a structure, with the order in which bits are transmitted on a network. The Ethernet, for example, conveys the bits of an octet in the order least–significant bit to most–significant bit.

The overall size of a sequence constructor type is the sum of the sizes of its component fields.

DRN allows one to describe a sequence field whose form depends on the contents of another field that occurs earlier in the same sequence. The earlier field, which determines the form of the later one, is called the *determinant*. A determinant must be an Integer, an UnsignedInteger, or an enumerated type data element. The specification of the later field will make reference to the determinant. This refere[illegible]s in the form of a series of period–separated field names, which refer to fields [illegible]n consecutively nested sequences. The manner of specifying a determinant is best illustrated with an example; one is presented in the next section.

A.6 Array type constructor

An *array* is another kind of type constructor. It specifies an ordered collection of data elements of the same type, and it is written:

```
array ( boundsList ) of typeSpec
```

The *boundsList* is a series of comma–separated items, specifying the dimensions of an array of data elements of type *typeSpec*. Each item of *boundsList* has one of these forms:

```
number
valueName
determinant
```

If the item is a number, or a name that has been associated with a constant value, then the corresponding dimension of the array is fixed at that value.

If the array is being specified as a field within a sequence, then a *determinant* can be used to define an array dimension that is variable. Here is an example of a sequence that includes an array of variable dimension:

```
sequence {
    size UnsignedInteger (8),
    data array (size) of Character (8)
}
```

In this example, the number of elements in the array `data` is determined by the contents of the `size` field, for any particular instance of the sequence. In the following example, the determinant refers to a field within a field:

```
sequence {
    parameters sequence {
        max Float (32),
        count UnsignedInteger (32)
    },
    data array (parameters.count) of Float (32)
}
```

Array elements are represented in order, with the last array subscript varying most frequently. Successive elements occupy consecutive octets. The overall size of an array is the size of an array element times the number of elements.

A.7 Choice type constructor

A *choice* is the third kind of type constructor. It describes a selection of alternate data elements, and the conditions under which each alternate will be represented.

A choice type constructor is written:

```
choice ( determinant ) of { whenList }
```

where *whenList* is a series of clauses separated by commas. Each clause has the form:

```
when ( choiceValue ) choiceName typeSpec
```

A choice may appear only as a field within a sequence. The *determinant* specifies a data element that occurs earlier in the same sequence according to the conventions described in the previous section on sequences.

Each clause specifies one alternate representation of the choice that corresponds to a particular value of the data element specified by the *determinant*. That value, *choiceValue*, may be specified in the clause as a number, as a name that has been associated with a constant value by a constant definition, or as a name that has been associated with a constant value by the definition of an enumerated type. The *typeSpec* describes the representation corresponding to that value. The *choiceName* is a name included as a convenient way of referring to the representation alternative. By

convention, such names begin with lower–case letters. Each must be unique among those of the same *whenList*.

There may be no duplicates among the values specified in clauses.

If, for any particular instance of the choice, the *fieldSpec* data element has a value not listed among the clauses, then the choice is not represented (i.e., it occupies no bits of the representation). Otherwise, the size of the choice is the size of whichever alternate data element it represents.

A.8 Bit alignment of data elements

Many computers require that certain data elements always be aligned in memory on a certain multiple of their basic byte or word size. Correspondingly, DRN imposes restrictions on the sizes of some data elements and the arrangement of sequence fields to ensure that certain alignment constraints will always apply.

Here we define these restrictions by associating a notional attribute, *alignment*, with each data element. A data element with an alignment attribute of **n** must start a multiple of **n** bits from the beginning of the PDU. The value of a data element's alignment attribute is determined as follows:

- If the data element is of a primitive type, and its size is 8, 16, 32, or 64 bits, then its alignment attribute is 8, 16, 32, or 64, respectively.
- If the data element is of a primitive type, and its size is not 8, 16, 32, or 64 bits, then its alignment attribute is 1.
- If the data element is a sequence then its alignment attribute is the maximum of the alignment attributes of the sequence's fields, and 8.
- If the data element is an array then its alignment attribute is that of its array element.
- If the data element is a choice then its alignment attribute is the maximum of the alignment attributes of its alternate components.

These restrictions are imposed on sequences:

- The size of a sequence, measured in bits, must be a multiple of its alignment attribute, and a multiple of 8.
- The bit offset of a field within a sequence must be a multiple of the field's alignment attribute.

- If a field has an alignment attribute of 1, then the following must hold:

$$\left\lfloor \frac{\text{bit offset of first bit of field}}{32} \right\rfloor = \left\lfloor \frac{\text{bit offset of last bit of field}}{32} \right\rfloor$$

This restriction is imposed on arrays:

- The size of an array element, measured in bits, must be both a multiple of its alignment attribute, and a multiple of 8 bits.

This restriction is imposed on choices:

- The size of choice must be a multiple of its alignment attribute.

APPENDIX B: OBJECT TYPE NUMBERING SCHEME

Objects in the simulated world include vehicles, buildings, bridges, projectiles, portions of fuel, plants, soldiers, etc. It is necessary, for the purposes of communication, to be able to describe an object in a succinct manner. This is done by using codes to refer to the types of objects, and by communicating these codes. This appendix describes the scheme according to which object type codes are defined. The following appendix lists the codes that are presently defined for specific types of objects.

An object type code is represented in 32 bits. The value 0 is reserved; it is used in particular cases to mean that an object type code is not applicable, or has not been defined.

The 32 bits are interpreted as a series of fields, each spanning some number of consecutive bits. The interpretation must be performed by examining the fields from left (most–significant bit) to right (least–significant bit). This is because, at each point in the series, fields to the left may determine the format of fields to the right.

The first field always occupies the leftmost 3 bits of the 32 bit code. This field is called the `domain` field. The values of this field, as presently defined, are:

0	Other (miscellaneous)
1	Vehicle (e.g., tank, submarine)
2	Munition (e.g., projectile, detonator, fuel, repair part)
3	Structure (e.g., building, bridge)
4	Life Form (e.g., tree, soldier, Martian)

The following subsections describe how the remaining 29 bits of the object type code are interpreted for the values of the `domain` field representing vehicles, munitions and life forms. Object type codes for other domains have not been defined.

Values of all fields are defined sequentially, starting from 1. 0 is reserved for "miscellaneous".

B.1 Vehicle type scheme

The domain of vehicles includes platforms that operate on land, in the air, on or below the sea, and in space. It includes both towed and self–propelled platforms. Guided missiles and rockets have been classified as munitions rather than vehicles.

The object type code describing a vehicle is divided into these fields, listed here from left to right:

`domain`	3 bits	contains the value 1, denoting a vehicle
`environment`	3 bits	describes the environment in which the vehicle operates (ground, air, water)
`class`	3 bits	organizes vehicles within a particular environment into broad classes (e.g., wheeled vs. tracked, fixed wing vs. rotary wing)
`country`	6 bits	describes the country to which the vehicle's design is attributed (e.g., USA, USSR)
`series`	6 bits	identifies a particular vehicle chassis (e.g., M1, M113)
`model`	6 bits	identifies a particular model of vehicle (e.g., M1, M1A1, M1AI Block II)
`function`	5 bits	identifies the function of the vehicle (e.g., main battle tank, armored personnel carrier, reconnaissance vehicle)

The value of the `class` field is interpreted within the context of a particular `environment` field value. For example, if the `environment` field is "air", the `class` field specifies whether the vehicle is a fixed wing or a rotary wing vehicle; if the `environment` field is "ground", `class` specifies wheeled or tracked.

The chart below shows the `environment` and `class` field values that are presently defined, and the relationship among them. Field values are indicated in parentheses.

Environment	Class
Air (1)	Fixed Wing (1) Lighter Than Air (2) Rotary Wing (3)

Ground (2)	Self–Propelled, Armored, Tracked (1) Self–Propelled, Armored, Wheeled (2) Self–Propelled, Unarmored, Tracked (3) Self–Propelled, Unarmored, Wheeled (4) Towed (5)
Space (3)	
Water (4)	Amphibious Warfare (1) Auxiliary (2) MaterialSupport (3) Mine Warfare (4) Submarine (5) Surface Combat (6)

The interpretation of the `country` field value is fixed, regardless of the `environment` and `class` fields. This interpretation is defined below, in the section entitled "Country Codes".

The `series` field is interpreted within the context of the `country`, `class` and function fields, and the `model` field is interpreted within the context of those fields plus the `series` field. Appendix C lists currently defined vehicle type codes, along with their `series` and `model` field values.

The interpretation of the `function` field is dependent only on the `environment` field. For air vehicles, the values of the `function` field are:

0	Miscellaneous
1	Air Combat
2	Ground Attack
3	Reconnaissance
4	Bomber

For ground vehicles, the values of the `function` field are:

0	Miscellaneous
1	Anti–Aircraft Gun or Surface–to–Air Missile

2 Armored Personnel Carrier

3 Command Post

4 Howitzer or Anti–Tank Gun

5 Mortar

6 Multiple Rocket Launcher

7 Reconnaissance

8 Recovery

9 Supply Truck

10 Tank Destroyer

11 Tank, Light

12 Tank, Main Battle

13 Combat Engineering

14 Surface to Surface Missile Launcher

For vehicles of other environments, one `function` field value is presently defined:

0 Miscellaneous

B.2 Munition type scheme

The domain of munitions includes military supplies other than vehicles and munitions, petroleum, oil, and lubricants; repair parts; medical supplies; etc.

The object type code describing a munition has these as its leftmost fields:

`domain`	3 bits	contains the value 2, denoting a munition
`class`	4 bits	describes the class of munition: projectile, fuel, repair part, etc.

The values of the `class` field, as presently defined, are:

0 Miscellaneous

1 Detonator

2 Missile

3 Petroleum, Oil, and Lubricants

4	Projectile
5	Propellent
6	Bomb
7	Mine

Of these classes of munition, the detonator, projectile, and propellent classes are treated in a similar manner. This is described in the next subsection, entitled "Ammunition type scheme". Object type codes for missiles and bombs are described separately in two following subsections. Object type code schemes for the other classes of munitions have not been defined.

B.2.1 Ammunition type scheme

For the classes detonator, projectile, and propellent, the entire object type code is divided into these fields:

`domain`	3 bits	contains the value 2, denoting munition
`class`	4 bits	describes the class of munition: projectile, propellent, or detonator
`caliber`	5 bits	describes the caliber of munition
`subclass`	4 bits	organizes munitions within a class into smaller subclasses (e.g., time detonator vs. percussion detonator, or training projectile vs. high explosive projectile)
`country`	6 bits	describes the country to which the ammunition's design is attributed (e.g., USA, USSR)
`series`	5 bits	identifies a particular series of munition
`model`	5 bits	identifies a particular model of munition

The value of the `subclass` field is interpreted within the context of a particular `class` field value, but independently of the `caliber` field value. The chart below shows the relationship between `class` and `subclass` field values, for those subclasses that are presently defined. Field values are indicated in parentheses.

Class	Subclass

Detonator (1)	Percussion (1) Proximity (2) Time (3)
Projectile (3)	Biological (1) Bomblets (2) Chemical (3) High Explosive (4) High Explosive, Plastic (5) High Explosive, Incendiary (6) Illumination (7) Kinetic (8) Nuclear (9) Practice (10) Shaped Charge (11) Smoke (12)
Propellent (4)	Bagged (1) Canistered (2)

The `caliber` field has a common interpretation for all three classes of ammunition—detonator, projectile, and propellent. Its values each represent a range of ammunition caliber:

0	caliber not applicable
1	caliber > 0 mm, but ≤10 mm
2	caliber > 10 mm, but ≤ 20 mm
3	caliber > 20 mm, but ≤ 30 mm
...	
31	caliber > 300 mm

The interpretation of the `country` field value is fixed, regardless of the class of ammunition or of the content of the `subclass` field. This interpretation is defined below, in the section entitled "Country codes".

The `series` field is interpreted within the context of the `country`, `subclass`, and `caliber` fields, and the `model` field is interpreted within the context of those fields plus the `series` field. Appendix C lists currently defined munition type codes, along with their `series` and `model` field values.

B.2.2 Missile type scheme

For the missile class of munitions, the entire object type code is divided into these fields:

`domain`	3 bits	contains the value 2, denoting munition
`class`	4 bits	contains the value 2, denoting missile
`target`	5 bits	describes the intended target of the missile
`warhead`	4 bits	indicates the type of warhead that the missile carries
`country`	6 bits	describes the country to which the ammunition's design is attributed (e.g., USA, USSR)
`series`	5 bits	identifies a particular series of munition
`model`	5 bits	identifies a particular model of munition

The `target` field indicates the type of target that the missile is intended to be used against. The values presently defined for this field are:

0	Miscellaneous
1	Anti–Aircraft
2	Anti–Armor
3	Anti–Missile
4	Anti–Radar
5	Anti–Satellite
6	Anti–Ship
7	Anti–Submarine
8	Anit-Surface

The `warhead` field indicates the type of warhead carried by the missile. The values presently defined for this field are:

0	Miscellaneous
1	Biological
2	Bomblets
3	Chemical

4	High Explosive
5	High Explosive, Plastic
8	Kinetic
9	Nuclear
11	Shaped Charge

The interpretation of the `country` field value is fixed, regardless of the contents of the `target` and `warhead` fields. This interpretation is defined below, in the section entitled "Country codes".

The `series` field is interpreted within the context of the `country`, `warhead`, and `target` fields, and the `model` field is interpreted within the context of those fields plus the `series` field. Appendix C lists currently defined missile type codes, along with their `series` and `model` field values.

B.2.3 Bomb type code scheme

For the bomb class of munitions, the entire object type code is divided into these fields:

`domain`	3 bits	contains the value 2, denoting munition
`class`	4 bits	contains the value 6, denoting bomb
`weight`	5 bits	specifies the bomb's nominal weight
`subclass`	4 bits	organizes bombs into various subclasses (e.g., general purpose, cluster, demolition)
`country`	6 bits	describes the country to which the bomb's design is attributed (e.g., USA, USSR)
`series`	5 bits	identifies a particular series of bomb
`model`	5 bits	identifies a particular model of bomb

The `weight` field represents the nominal weight of the bomb, using the following encoding:

0	nominal weight not applicable
1	nominal weight > 0 lb., but ≤100 lb.

2 nominal weight > 100 lb., but ≤ 200 lb.

3 nominal weight > 200 lb., but ≤ 300 lb.

...

31 nominal weight > 3000 lb.

The value of the `subclass` field is interpreted independently of the `weight` field. It is one of these values:

0 Miscellaneous Bomb

1 General Purpose Bomb

The interpretation of the `country` field value is fixed, regardless of the contents of the `weight` and `subclass` fields. This interpretation is defined below, in the section entitled "Country codes".

The `series` field is interpreted within the context of the `country`, `subclass`, and `weight` fields, and the `model` field is interpreted within the context of those fields plus the `series` field. Appendix C lists currently defined bomb type codes, along with their `series` and `model` field values.

B.2.4 Mine type code scheme

For the mine class of munitions, the entire object type code is divided into these fields:

`domain`	3 bits	contains the value 2, denoting munition
`class`	4 bits	contains the value 7, denoting mine
`target`	5 bits	specifies the mine's nominal target
`environment`	4 bits	specifies the medium in which the mine is designed to operate
`country`	6 bits	describes the country to which the mine's design is attributed (e.g., USA, USSR)
`series`	5 bits	identifies a particular series of mine
`model`	5 bits	identifies a particular model of mine

The `target` field indicates the principal type of target that the mine is designed to be used against. The values presently defined for this field are:

0 Miscellaneous

1 Tank

2 Person

The `environment` field indicates the environment in which the mine is intended to operate. The values presently defined for this field are:

0 Miscellaneous

1 Land

2 Water

The interpretation of the `country` field value is fixed, regardless of the contents of the `target` and `environment` fields. This interpretation is defined below, in the section entitled "Country codes".

The `series` field is interpreted within the context of the `country`, `target`, and `environment` fields, and the `model` field is interpreted within the context of those fields plus the `series` field. Appendix C lists currently defined mine type codes, along with their `series` and `model` field values.

B.3 Life form type scheme

The domain of life forms includes any living organism.

The object type code describing a life form consists of the following fields:

`domain`	3 bits	contains the value 4, denoting a life form
`type`	29 bits	identifies the type of life form

Appendix C lists currently defined life forms.

B.4 Country codes

Object type codes for both vehicles and munitions include a field specifying the country of origin of the vehicle or munition type. This is used, for example, to subdivide types of tanks into those originating in the U.S.A., those originating in the U.S.S.R., etc.

The values of the country field, as presently defined, are:

0	Other
1	U.S.A.
2	U.S.S.R.
3	Germany

APPENDIX C: DEFINED OBJECT TYPE CODES

This appendix enumerates the object type codes that are presently defined for various types of vehicles and munitions.

Wherever a number appears in parentheses in this appendix, it is defining a value for a field of an object type code. The value 0 is defined for fields whose contents, in a particular case, are not presently used to distinguish among types of objects. For example, in the subsection below defining object type codes for U.S. vehicles, it is specified that an M109 howitzer is referred to by a `series` field value of 4, and a `model` field value of 0; no codes for distinct models of the M109 are presently defined.

C.1 Object type codes for vehicles

Each of the following subsections lists object type codes defined for vehicles of a particular country. Within each subsection, the lists are organized by other fields of the vehicle type code: `environment`, `class`, and `function`. All object type codes for vehicles have a `domain` field value of 1.

C.1.1 U.S. vehicles

All object type codes for U.S. vehicles have a `country` field value of 1. The following list defines valid `series` and `model` field values for particular values of the `environment`, class, and function fields.

Environment; Class	Series	Model	Function
Air; Fixed Wing	A–10 (1)	(0)	Ground Attack
	F-16 (2)	A (1)	Air Combat
		B (2)	Air Combat
		C (3)	Air Combat
		D (4)	Air Combat
	F-14 (3)	A (1)	Air Combat
		D (2)	Air Combat
Air; Rotary Wing	AH–64 (1)	(0)	Ground Attack
	OH-58D(2)	(0)	Air Reconnaissance
Ground; SP, Armored, Tracked	M1 Abrams (1)	(0)	Tank, Main Battle
	M2 Bradley (2)	M2 (1)	APC
		M3 (2)	Reconnaissance

	M113 (3)	M113A2 (1)	APC
		M577 (2)	Command Post
		M106A1 (3)	Mortar
	M109 155mm (4)	(0)	Howitzer
	M88 (5)	A1 (1)	Recovery
	ADATS (6)	(0)	Anti-Aircraft
	LOSAT (7)	(0)	Tank Destroyer
Ground; SP, Unarmored, Wheeled	M35 2.5 ton (1)	M35A2 (1)	Supply Truck
	HEMTT (2)	M977 (1)	Supply Truck
		M978 (2)	Supply Truck
Ground; Towed,	M57 Mine layer (1)	(0)	Combat Engineering
	M128 GEMSS (2)	(0)	Combat Engineering
	M58A1 (3)	(0)	Combat Engineering
	Towed pallet (4)	(0)	Supply Truck

C.1.2 Soviet vehicles

All object type codes for Soviet vehicles have a `country` field value of 2. The following list defines valid `series` and `model` field values for particular values of the `environment`, `class` and function fields.

Environment; Class	Series	Model	Function
Air; Fixed Wing	Su-25 (1)	(0)	Ground Attack
	MiG-23 C (2)	(0)	Air Combat
	MiG-27 D (3)	(0)	Air Combat
	MiG-21 C (4)	(0)	Air Combat
	MiG-25 A (5)	(0)	Air Combat
	MiG-29 (6)	(0)	Air Combat
	MiG-31 (7)	(0)	Air Combat
Air; Rotary Wing	Mi-24 (1)	(0)	Ground Attack
	Mi-28 (2)	(0)	Ground Attack
	Mi-8 (3)	(0)	Ground Attack
	Mi-17 (4)	(0)	Ground Attack

Ground; SP, Armored, Tracked	T–72 (1)	M (1)	Tank, Main Battle
	BREM-1 (1)	(2)	Recovery
	BMP-1 (2)	(1)	APC
		K (2)	Command Post
	2S1 122mm (3)	(0)	Howitzer
	BMP-2 (4)	(0)	APC
	ZSU-23/4M (5)	(0)	Anit-Aireraft
	ACRV (6)	(0)	Command Post
	T-80 (7)	(0)	Tank, Main Battle
	BRM (8)	(0)	Ground Reconnaissance
	T-64 (9)	(0)	Tank, Main Battle
	T-62 (10)	(0)	Tank, Main Battle
	T-55 (11)	(0)	Tank, Main Battle
	T-54 (12)	(0)	Tank, Main Battle
Ground; SP, Armored, Wheeled	BDRM (1)	(0)	APC
	BRDM-2 (9)	(0)	Ground Reconnaissanee
Ground; SP, Unarmored, Wheeled	Mine layer (0)	(0)	Combat Engineering
	GAZ–66 (1)	(0)	Supply Truck
	BM-21 (1)	(1)	Roeket Launeher
	Ural-375 (2)	Cargo (1)	Supply Truck
		Fuel (2)	Supply Truck
	BTR-80 (3)	(0)	APC
Ground; Towed	PMR-3 (0)	(0)	Combat Engineering
	MICLIC (1)	(0)	Combat Engineering
	155mm (1)	(0)	Howitzer
	M-1943 (1)	(0)	Mortar
	Towed pallet (1)	(0)	Supply Truck

C.1.2 German vehicles

All object type codes for German vehicles have a `country` field value of 3. The following list defines valid `series` and `model` field values for particular values of the `environment`, `class` and function fields.

Environment; Class	Series	Model	Function
Ground; SP, Armored, Tracked	Leo-2 (1)	(0)	Tank, Main Battle
	Marder (2)	(0)	APC

C.2 Object type codes for munitions

There is presently one munition type code defined for fuel. It has a `domain` field value of 2 (denoting a munition) and a `class` field value of 3 (denoting POL); the remaining fields are zero.

Each of the following subsections lists object type codes defined for ammunition or missiles of a particular country. Within each subsection, the lists are organized by other fields of the munition type code: `class`, `caliber`, and `subclass` in the case of ammunition; or `target` and `warhead` in the case of missiles. All object type codes for munitions have a `domain` field value of 2.

C.2.1 U.S. ammunition

All object type codes for U.S. ammunitions have a `country` field value of 1. The following list defines valid `series` and `model` field values for particular values of the `class, caliber` and subclass fields.

Class; Caliber; Subclass	Series	Model
Detonator; Mine; Proximity	M603 (1)	(0)
Detonator; 107mm; Percussion	M557 (1)	(0)
Detonator; 107mm; Proximity	M513 (1)	(0)
Detonator; 155mm; Percussion	M739 (1)	(0)
Detonator; 155mm; Timed	M728 (1)	(0)
Detonator; Bomb; Percussion	M904(1)	(0)
Projectile; 5.56 mm; Kinetic	M855 (1)	(1)
	M856 (1)	(2)
Projectile; 25mm; Kinetic	M791 (1)	(0)
Projectile; 25mm; High Explosive	M792 (1)	(0)
Projectile; 105mm; Kinetic	M392 (1)	M392A2 (1)

Projectile; 105mm; Shaped Charge	M456 (1)	M456A1 (1)
Projectile; 107mm; High Explosive	M329 (1)	(0)
Projectile; 155mm; High Explosive	M107 (1)	(0)

C.2.2 U.S. missiles

All object type codes for U.S. missiles have a `country` field value of 1. The following list defines valid `series` and `model` field values for particular values of the `target` and `warhead` fields.

Target; Warhead	Series	Model
Anti–armor; Shaped Charge	TOW (1)	(0)
	M47 Dragon (2)	(0)
	Hellfire (3)	(0)
	Maverick (4)	(0)
Anti–air; High Explosive	Sidewinder (1)	(0)
	ADATS (2)	(0)
	Stinger (3)	(0)

C.2.3 U.S. bombs

All object type codes for U.S. missiles have a `country` field value of 1. The following list defines valid `series` and `model` field values for particular values of the `weight` and `subclass` fields.

Weight; Subclass	Series	Model
500 lb.; General Purpose	Mk82 (1)	(0)

C.2.3 U.S. mines

All object type codes for U.S. mines have a `country` field value of 1. The following list defines valid `series` and `model` field values for particular values of the `target` and `environment` fields.

Target; Environment	Series	Model
Tank; Land	M15 (1)	(1)
	M19 (2)	(1)
	M21 (3)	(1)

	M741 (4)	(1)
	M718 (5)	(1)
	M75 (6)	(1)
Person; Land	M14 (1)	(1)
	M18A1 (2)	(1)
	M16A2 (3)	(1)
	M731 (4)	(1)
	M692 (5)	(1)
	M74 (6)	(1)

C.3 Object type codes for life forms

All type codes for life forms have a `domain` field value of 4. There are two life forms defined. The `type` field value of 1 represents a U.S. infantryman. The `type` field value of 2 represents a Soviet infantryman.

APPENDIX D: VEHICLE–SPECIFIC PROTOCOL

This appendix defines those aspects of the SIMNET protocols that are specific to a particular type of vehicle simulator.

D.1 SIMNET M1 Abrams main battle tank

This section defines those aspects of the SIMNET protocols that are specific to the SIMNET M1 Abrams Main Battle Tank simulator [1].

D.1.1 Repairs

The SIMNET M1 simulator recognizes the following values of the Repair Type data element:

```
type SIMNET_M1_RepairType enum (16) {
    m1ReplaceAlternator (1),
    m1ReplaceBattery (2),
    m1ReplaceEngineOilFilter (3),
    m1ReplaceTransOilFilter (4),
    m1ReplacePrimaryFuelFilter (5),
    m1ReplacePilotRelayStarter (6),
    m1ReplacePowerPack (7),
    m1RepairServiceBrake (8),
    m1RepairParkingBrake (9),
    m1RepairTurretTraverseDrive (10),
    m1RepairTurretMountInterface (11),
    m1RepairGunElevationDrive (12),
    m1RepairStabSystem (13),
    m1RepairLRF (14),
    m1RepairFuelTransferPump (15),
    m1RepairGPS (16),
    m1RepairCdrExtGPS (17)
}
```

These values are interpreted as follows:

`m1ReplaceAlternator`	replace alternator
`m1ReplaceBattery`	replace battery
`m1ReplaceEngineOilFilter`	replace engine oil filter
`m1ReplaceTransOilFilter`	replace transmission oil filter

m1ReplacePrimaryFuelFilter	replace primary fuel filter
m1ReplacePilotRelayStarter	replace pilot relay and starter
m1ReplacePowerPack	replace engine and transmission
m1RepairServiceBrake	repair service brake
m1RepairParkingBrake	repair parking brake
m1RepairTurretTraverseDrive	repair turret traverse drive
m1RepairTurretMountInterface	repair turret mount interface
m1RepairGunElevationDrive	repair gun elevation drive
m1RepairStabSystem	repair gun stabilization system
m1RepairLRF	repair laser rangefinder
m1RepairFuelTransferPump	repair fuel transfer pump
m1RepairGPS	repair gunner's primary sight (GPS)
m1RepairCdrExtGPS	repair commander's extension to the GPS

D.1.2 Vehicle specific status

The following structure describes M1–specific status information, including the quantities of fuel and ammunition stowed in various compartments of the M1:

```
type SIMNET_M1_Status sequence (
    enginePower          Float (32),
    battery              Float (32),
    frontLeftFuel        Float (32),
    frontRightFuel       Float (32),
    rearFuel             Float (32),
    apdsReadyAmmo        UnsignedInteger (8),
    apdsSemiReadyAmmo    UnsignedInteger (8),
    apdsHullTurretFloorAmmo UnsignedInteger (8),
    heatReadyAmmo        UnsignedInteger (8),
    heatSemiReadyAmmo    UnsignedInteger (8),
    heatHullTurretFloorAmmo UnsignedInteger (8),
                         unused (208)
}
```

The `enginePower` field specifies the fraction of full power the vehicle's engine is able to produce. The `voltage` of the vehicle's battery (in volts) is specified by the `battery`

field, and the quantity of fuel in each tank (in gallons) is specified by the `frontLeftFuel`, `frontRightFuel`, and `rearFuel` fields.

The vehicle carries ammunition of two types (APDS and HEAT) distributed among three places (the ready rack, the semi–ready rack, and the hull or turret floor stowage). The last six fields of the SIMNET_M1_Status data element specify the number of rounds of each type in each of these places.

D.2 SIMNET M2/3 Bradley fighting vehicle

This section defines those aspects of the SIMNET protocols that are specific to the SIMNET M2/3 Bradley Fighting Vehicle simulator [2].

D.2.1 Repairs

The SIMNET M2/3 simulator recognizes the following values of the Repair Type data element:

```
type SIMNET_M2_RepairType enum (16) {
    m2ReplaceGenerator (1),
    m2ReplaceBattery (2),
    m2ReplaceEngine (3),
    m2ReplaceStarter (4),
    m2ReplaceEngineFuelFilter (5),
    m2ReplaceAirCleaner (6),
    m2ReplaceTransmission (7),
    m2ReplaceTurretDistBox (8),
    m2ReplaceGunnersCtlHandle (9),
    m2ReplaceCmdrsCtlHandle (10),
    m2ReplaceTurretPositionInd (11),
    m2ReplaceTurretSlopeInd (12),
    m2RepairCoolantLeak (13),
    m2RepairTransOilLeak (14),
    m2RepairEngineOilLeak (15),
    m2RepairServiceBrake (16),
    m2RepairParkingBrake (17),
    m2RepairGunnersSight (18),
    m2RepairGunElevationDrive (19),
    m2RepairTurretTraverseDrive (20),
    m2RepairCannonMountInterface (21),
    m2RepairTOWLauncher (22),
```

```
        m2RepairIntercom (23)
    }
```

These values are interpreted as follows:

m2ReplaceGenerator	replace generator
m2ReplaceBattery	replace battery
m2ReplaceEngine	replace engine
m2ReplaceStarter	replace engine starter
m2ReplaceEngineFuelFilter	replace engine fuel filter
m2ReplaceAirCleaner	replace air cleaner
m2ReplaceTransmission	replace transmission
m2ReplaceTurretDistBox	replace turret electrical distribution box
m2ReplaceGunnersCtlHandle	replace gunner's control handle
m2ReplaceCmdrsCtlHandle	replace commander's control handle
m2ReplaceTurretPositionInd	replace turret position indicator
m2ReplaceTurretSlopeInd	replace turret slope indicator
m2RepairCoolantLeak	repair engine coolant leak
m2RepairTransOilLeak	repair transmission oil leak
m2RepairEngineOilLeak	repair engine oil leak
m2RepairServiceBrake	repair service brake
m2RepairParkingBrake	repair parking brake
m2RepairGunnersSight	repair gunner's sight
m2RepairGunElevationDrive	repair gun elevation drive
m2RepairTurretTraverseDrive	repair turret traverse drive
m2RepairCannonMountInterface	repair cannon mount interface
m2RepairTOWLauncher	repair TOW missile launcher
m2RepairIntercom	repair crew intercom

D.2.2 Vehicle specific status

The following structure describes M2/3–specific status information, including the quantities of fuel and ammunition stowed in various compartments of the vehicle:

```
type SIMNET_M2_Status sequence {
    enginePower          Float (32),
    hullBattery          Float (32),
    turretEmergencyBattery Float (32),
    topFuel              Float (32),
    bottomFuel           Float (32),
    apCanAmmo            UnsignedInteger (8),
    apCanAmmoType        UnsignedInteger (8),
    heCanAmmo            UnsignedInteger (8),
    heCanAmmoType        UnsignedInteger (8),
    apStowedAmmo         UnsignedInteger (16),
    heiStowedAmmo        UnsignedInteger (16),
    towStowedAmmo        UnsignedInteger (4),
    dragonStowedAmmo     UnsignedInteger (4),
    tow1Loaded           UnsignedInteger (1),
    tow2Loaded           UnsignedInteger (1),
    towLauncherUp        UnsignedInteger (1),
    m3Configuration      UnsignedInteger (1),
    rampDown             UnsignedInteger (1),
                         unused (179)
}
```

The `enginePower` field specifies the fraction of full power the vehicle's engine is able to produce. The voltage of the vehicle's two batteries (in volts) is specified by the `hullBattery` and `turretEmergencyBattery` fields, and the quantity of fuel in its two tanks (in gallons) is specified by the `topFuel` and `bottomFuel` fields.

Several fields are used to describe the vehicle's ammunition load as modeled by the M2/3 simulator. Twenty–five millimeter rounds are stored in three places within the vehicle: in a bin called the *AP can* attached to the turret, in a similar bin called the *HE can*, and in stowage compartments within the floor of the vehicle. Although the two cans are labelled AP and HE, they do not necessarily contain just APDS and HEI rounds respectively. Ammunition is loaded into the cans in strings of 30 rounds, and either bin may contain a mix of both APDS and HEI strings.

The SIMNET_M2_Status data element describes the quantity and mix of ammunition in each can using the following scheme. The `apCanAmmo` and `heCanAmmo` fields indicate

how many rounds of either type are in each of the two cans. The `apCanAmmoType` and `heCanAmmoType` fields indicate what types of ammunition the cans contain. This type information is encoded as a series of bits, with a single bit corresponding to each string in the series of ammunition strings contained in a can. If a bit is 0, the corresponding string contains APDS ammunition; if it is 1, the string contains HEI ammunition. The low–order bit of the 8–bit field corresponds to the first string of ammunition leading into the gun breach; this string may contain fewer than 30 rounds if some have been fired. Higher–order bits represent successive strings in the series, each of 30 rounds. If a can contains 210 rounds or less (seven strings or less), then some highest–order bit(s) of its type field are unused, and they should be 0.

The `apdsStowedAmmo`, `heiStowedAmmo`, `towStowedAmmo`, and `dragonStowedAmmo` fields specify the number of rounds of various types of ammunition stowed in the vehicle's floor compartments. The remaining bit fields of the SIMNET_M2_Status data element indicate whether each of the two TOW missile launchers is loaded, whether the launcher is raised or lowered, whether the simulated vehicle is the M2 or the M3 variant of the Bradley Fighting Vehicle, and whether the vehicle's rear ramp is raised or lowered.

APPENDIX E: ETHERNET IMPLEMENTATION

This appendix specifies how the data link layer services required by the SIMNET association protocol are provided by either of the two popular Ethernet standards. The preferred standard is IEEE 802.3 [12] (aka ISO 8802/3). Use of the older Ethernet Version 2.0 [11] standard is discouraged. Both of these local area networks meet all of the requirements described in chapter 4.

E.1 Overview

A full description of Ethernet may be found in the referenced documents [11] [12]. What follows is a brief summary of this network. An Ethernet provides a single physical channel operating at a fixed data rate of 10 Megabits per second. Ethernet may carry datagrams ranging in size from 368 to 12,000 bits. Several types of physical media are supported, but shielded coaxial cable is perhaps the most common.

We have modeled the behavior of an Ethernet supporting a distributed simulation of 500 simulators, each producing updates at the rate of five per second (somewhat more than the typical rate actually measured), and found that the Ethernet is easily capable of carrying the resulting network traffic with negligible delay. This model is described in a separate report, *SIMNET Ethernet Performance* [13].

E.2 Use of Ethernet Addresses

Ethernet supports 48-bit source and destination addresses. The association protocol places no requirements on the Ethernet source address. All association protocol PDUs are transmitted using a multicast destination address. The multicast address is formed from the multicast-group and protocol-identifier parameters of the association protocol datagram and transaction services as follows:

- The first transmitted bit of the destination address is 1, identifying it as a logical or group address.
- The second transmitted bit of the destination address is 1, identifying it as a locally administered address.
- The third through 32nd bits of the destination address are 0.
- The 33rd through 40th bits of the destination address are the association protocol multicast-group number, transmitted on the Ethernet in sequence from low–order bit to high–order bit.

- The 41st through 48th bits of the destination address are the association protocol protocol-identifier number, transmitted on the Ethernet in sequence from low–order bit to high–order bit.

Note that an Ethernet transmits data least significant bit first. Note also that the protocol-identifier parameter is carried by the `userProtocol` field of the Association PDU(see §6.5.1). It is not to be confused with the `kind` field of the same PDU or the SIMNET Association Protocol Identifier described below.

An Ethernet datagram may contain one or more association PDUs. All PDUs in a datagram must pertain to the same association protocol multicast-group and association protocol protocol-identifier.

E.3 SIMNET Association Protocol Identifier

Ethernet Version 2.0 and and IEEE 802.3 support different mechanisms for specifying the kind of higher level protocol which is carried by a data link layer PDU. The following sections describe how the SIMNET association protocol is identified for each of the two Ethernet standards.

E.3.1 Ethernet Version 2.0

The Ethernet Version 2.0 specification defines datagrams that include a 16–bit type field. In accordance with this specification, a datagram containing SIMNET association protocol information should have a type field whose value is 21,000 decimal or 5208 hexadecimal [16]. That value labels all datagrams containing SIMNET association PDUs and distinguishes them from any other types of datagrams traversing an Ethernet Version 2.0 network. The essential subset of Ethernet Version 2.0 Data Link Layer PDU is illustrated in figure E-1.

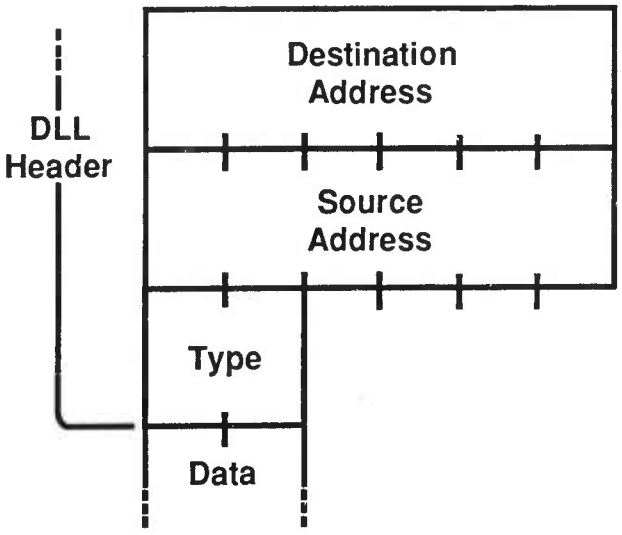


Figure E-1: Ethernet Version 2.0 Data Link Layer header.

E.3.2 IEEE 802.3

IEEE 802.3 splits the data link layer into two sublayers, the Media Access Control (MAC) sublayer and the Logical Link Control (LLC) sublayer. These are illustrated in figure E-2. The Media Access Control PDU is identical to the Ethernet Version 2.0 Data Link Layer PDU, except that the Ethernet Version 2.0 "Type" field has become a "Length" field. The "Length" field specifies the number of octets of data that follow.

The Logical Link Control PDU is encapsulated within the MAC PDU. The length in octets of the LLC PDU, including all headers, is the value of the length field of the MAC PDU. The Logical Link Control PDU contains four fields: DSAP, SSAP, Control, and Information. The Control field contains the value 3, indicating the LLC "Unnumbered Information" command (see [14]). The DSAP and SSAP specify the LLC's destination and source service access points. Because DSAP and SSAP are only eight bit fields, only a very limited number of global SAPs are permitted. Consequently, Project 802 has developed the "sub-network access protocol" (SNAP)[15]. SNAP allows a much larger number of protocols to be supported. Use of the SNAP protocol is indicated by DSAP and SSAP hexadecimal values of AA. When this is the case, the LLC Information field contains a SNAP PDU.

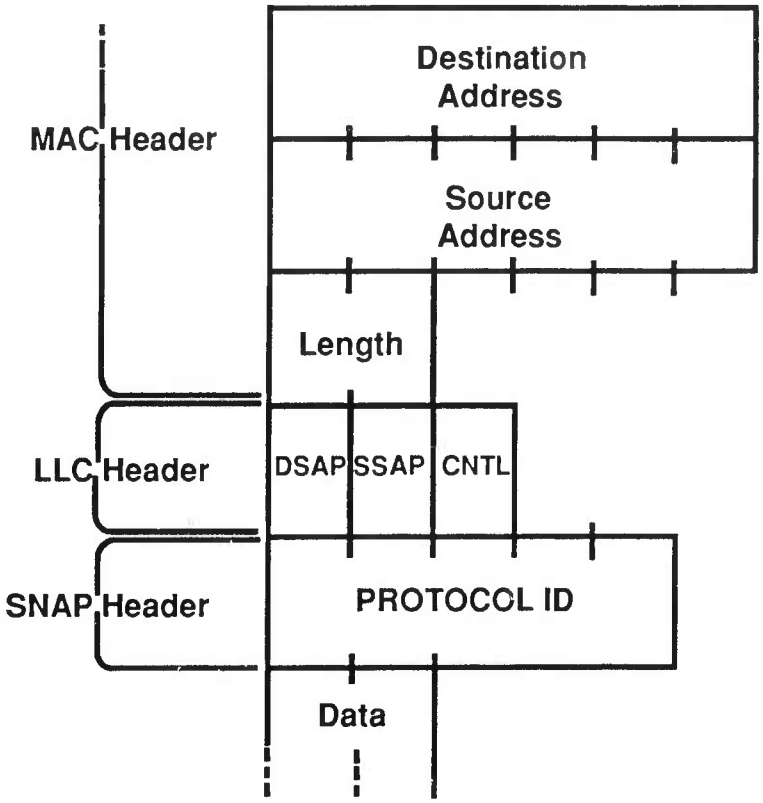


Figure E-2: IEEE 802 Ethernet Data Link Layer: Media Access Control, Logical Link Control and SNAP headers.

The SNAP PDU contains two fields: Protocol Identification and Protocol Data. The SIMNET association protocol has been assigned the unique protocol identifier 08-00-08-52-08 (hexadecimal). When this value in found in the Protocol Identification field of the SNAP, the SNAP Protocol Data field contains a SIMNET Association PDU.

APPENDIX F: TIMERS AND COUNTERS

The following timers are used in defining the SIMNET protocols:

Timer	Seconds	Description
`mineFieldTime`	30	The maximum time between any two Mine Field PDUs issued by any mine field.
`mineFieldTimeOut`	66	How long a simulator waits without receiving a Mine Field PDU describing a particular mine field before concluding that that mine field no longer exists.
`resupplyTime`	60	How long a supplier waits for a response to a Resupply Offer PDU.
`serviceCancelTime`	12	How long a supplier waits without receiving a Service Request PDU from a receiver before concluding that the receiver is no longer requesting service.
`serviceRequestTime`	5	The period between successive Service Request PDUs issued by a simulator requesting service for its vehicle.
`simulationStatusTime`	300	The maximum time between successive Simulation Status PDUs issued by a simulator.
`statusQueryTime`	5	The period a simulator waits to receive responses to a Status Query PDU issued using the datagram service.
`transactionCacheTime`	10	The period a cached transaction response is retained by an association entity.
`transactionRetryTime`	3	The period between successive transmissions of a Request APDU by an association entity.
`vehicleAppearanceTime`	5	The maximum time between successive Vehicle Appearance PDUs issued for any vehicle.

vehicleDisappearanceTime	12	How long a simulator waits without receiving a Vehicle Appearance PDU describing a particular vehicle before concluding that that vehicle no longer exists.
vehicleStatusTime	30	The maximum time between successive Vehicle Status PDUs issued by a simulator.

The following counters are used in defining the SIMNET protocols:

Counter	Repetitions	Description
transactionRetryCount	3	The maximum number of times a Request APDU is transmitted by an activation entity for one transaction.