NISTIR 6116

NGIS SIM Specification

William G. Rippey
John L. Michaloski
Martin Herman
Sandor Szabo
Intelligent Systems Division

William DeWys
Hughes Aircraft

Nathaniel Frampton
Real Time Development Corporation

Herbert Lau
Eun Soo Lee
Applied Precision Incorporated

John Rose
ExtrudeHone

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Bldg. 220 Rm. B124
Gaithersburg, MD 20899-0001

January 1998

NIST

NISTIR 6116

NGIS SIM Specification

William G. Rippey
John L. Michaloski
Martin Herman
Sandor Szabo
Intelligent Systems Division

William DeWys
Hughes Aircraft

Nathaniel Frampton
Real Time Development Corporation

Herbert Lau
Eun Soo Lee
Applied Precision Incorporated

John Rose
ExtrudeHone

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Bldg. 220 Rm. B124
Gaithersburg, MD 20899-0001

January 1998

NGIS SIM Specification

Draft – January 1998

Next Generation Inspection System Project
Sponsored by National Center for Manufacturing Sciences

Table of Contents

Foreword

Note on this document version – This NIST document is a draft version of the Sensor Interface Module Specification. As of December 1997 the specification has had about one year of development and testing. The Working Group is continuing to revise the document based on comments from NGIS members who are implementing software that conforms to the draft specification. The Working Group decided to publish this draft version as a NIST document to help distribution for wider review. The final version of the specification may differ from this document.

This document contains changes made as a result of the 18-Jan-98 Working Group meeting.

I. Introduction

1. Purpose of This Document

This document is an interface specification that addresses the integration of commercial probe hardware and software with commercial controllers. The application is control of coordinate measurement machines (CMMs) and numerically controlled (NC) machine tools to perform part inspection.

The document defines a standard hardware component, the Sensor Interface Module (SIM), and an application programming interface (API) to the SIM. The focus of this specification is on touch-trigger probes and scanning probes, but does not preclude applicability to other types. The people who are working on this specification form the SIM API Working Group. NIST staff serve on the Working Group and have acted as editors of the specification document.

Our premise is that the SIM will allow part inspection probes to be easily integrated into control systems for CMMs and for NC machine tools. This specification will be used by probe and controller suppliers. This specification provides for the capability to integrate a new probe at controller run time, without the need to recompile or reconfigure the controller software. Our goal is to provide sufficient guidance on technical properties of the SIM such that independently developed probe and controller products will be compatible right out of their "shrink-wrapped" packages. This document is the first product of the Working Group; enhancements will be considered in subsequent revisions.

An on-line version of the specification will be available through the Wide World Web, through the web site for the Intelligent Systems Division of NIST, http://isd.cme.nist.gov/info/ngisAPI.

2. The Next Generation Inspection System Project

This document is being developed as part of the Next Generation Inspection System - Phase II Project (NGIS II). NGIS II is sponsored by the National Center for Manufacturing Sciences (NCMS) and by its members. Members of the project are Advanced Technology and Research Corp., Applied Precision Inc (API), ExtudeHone, Ford Motor Company, General Motors, Hughes, ICAMP, NIST, Pratt and Whitney, and Sensor Adaptive Machines Inc (SAMI). The purpose of the project is to perform cooperative research and development to improve current industrial inspection techniques. Some members receive NCMS funds to supplement their efforts.

The original NGIS project was begun in November 1992 under NCMS management. Activities were aimed at defining inspection needs, and developing sensors for faster inspection methods. The next phase of the original effort was called NGIS II and was begun in 1997, assisted by new funding from NCMS.

The specification is being written by a Working Group consisting of staff from several NGIS II member companies. NIST staff are serving on the Working Group and are the editors of this document.

Members of the SIM Specification Working Group are: Bill DeWys (Hughes), Nat Frampton (Real Time Development Corp.), Marty Herman (NIST), Herbert Lau (API), Eun Soo Lee (API), John Michaloski (NIST), Bill Rippey (NIST), John Rose (ExtrudeHone), Sandor Szabo (NIST).

3. Document Overview

This specification contains the following parts:

- I. Introduction
- II. SIM Requirements Specification
- III. SIM Object Model
- IV. SIM Specification Sheet
- Appendix A. SIM Conformance Testing Specification
- Appendix B. Discussion of SIM Communications Technologies
- Appendix C. SIM Dynamic Linked Library (DLL) API Specification (sim.h)
- Appendix D. SIM State Diagram. This was added as a result of a working group meeting on 18-Jan-98. It has not been reviewed in detail by the group but is included for discussion.

II. SIM Requirements Specification

1. Introduction

1.1 Purpose

The purpose of defining a Sensor Interface Module is to standardize the characteristics and interfaces of certain hardware and software probe subsystems so that those subsystems can be developed as commercial products. These products will be interchangeable among the controllers of different vendors.

The audience consists of controller developers, probe developers, and control system integrators who take components from different suppliers and build complete control systems. The control systems are used for NC machine tools and CMMs for inspection of manufactured parts.

1.2 Scope

This document describes two components defined within NGIS II:

1. **Sensor Interface Module** (SIM) - a hardware card that plugs into the Industry Standard Architecture (ISA) computer bus and supplies an interface between a sensor and a machine control system.
2. SIM **Application Programming Interface** (API) - a software specification for a function-call interface between a machine controller executive and a SIM.

Figure II-1 shows a simplified view of a controller for a CMM or NC machine tool. A controller comprises its executive, a Motion Interface Module (MIM) and one or more SIMs. Statements in an inspection plan direct machine actions to inspect parts. The executive reads the plan and issues commands to the MIM and SIMs, and accesses data from them. The MIM accepts motion commands and generates commands to individual machine actuators. It also reads machine axis sensors and makes the data available to the executive. SIMs accept commands to configure themselves and to read data from inspection probes.

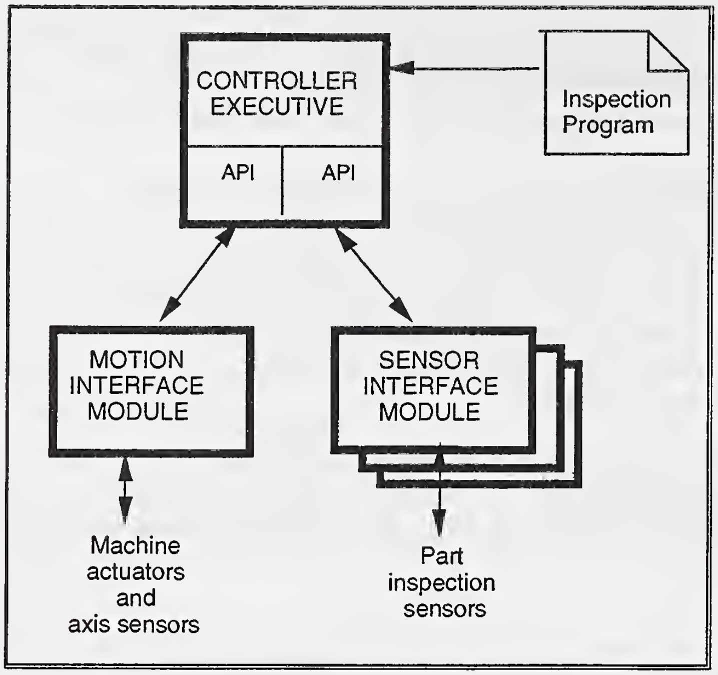


Figure II-1. NGIS Machine Controller Application Context

This document covers probes that produce proximity data (distance from a sensor to a point on an object) and conditioned measure of surface finish. The probes can be used for single-point measurements as well as scanning measurements. Specific probes in NGIS II are Accuprobe, SmartProx, CapScan, and Midas[*]. More complex probes such as vision systems may be addressed in a later specification. Machine control applications not addressed directly in this version are the servoing of machine axes using probe data, use of video probes, reverse engineering, fusion of probe data, and use of diagnostics.

This specification does not cover interfaces between sensors and signal conditioning hardware. These interfaces can be manufacturer-specific.

This specification is intended to be used by probe and controller suppliers. It provides for the capability to integrate a new probe while a controller is running, without the need to recompile or reconfigure the controller software. Our goal is to provide sufficient guidance on technical properties of probe components such that independently developed probe and controller products will be compatible right out of their "shrink-wrapped" packages.

If this specification is accepted by controller vendors, potential benefits include:

- little engineering effort will be needed to interface probes to existing controllers.
- controller vendors will be able to offer customers a wider range of integrated probes

[*] NOTE: Any trade names used in this report are given solely to provide complete identification of the equipment used. Such identification of products neither constitutes nor implies endorsement of the products or their manufacturers by NIST.

- probe vendors can develop new technology knowing their products will be compatible with compliant controllers. This may encourage entrepreneurial development of more probe technology, thereby increasing competition.
- CMM and NC users will be able to take advantage of a wider range of probes, without relying on controller vendors to integrate probes on a case-by-case basis, or without having to buy new controllers.
- users will have access to better measurement systems, at lower cost.

1.3 Definitions, Acronyms, Abbreviations

- *API* - "application programming interface", the protocol used to interact with a software module (e.g. math library, DLL, or OLE server).
- *CMM* - "coordinate measuring machine".
- *CNC* – “computer numerical control”.
- *COM* – “Component object model”. “Microsoft’s OLE object-oriented programming model that defines how objects interact within a single process or between processes”.[6]
- *conversion time* - the total time elapsed for a SIM to sample, digitize, optionally convert a sensor signal to engineering units, and store the data in its output queue.
- *DMIS* – “Dimensional Measuring Interface Standard”, ANSI/CAM-I 101-1995.
- *displacement-measuring probe* - "a probe that gives a signal proportional to a displacement of the probe from its free position". [3]
- *DLL* – “dynamic-link library”, also written ‘dll’. Microsoft technology, “...a file that contains functions compiled, linked and stored separately from the processes that use them”. [6]
- *FIFO* - "first in first out", a queue of elements where all insertions are made at one end and all removals and accesses at the other.
- *host* - a computer providing primary services to other computers or devices (peripherals, slave card, etc.).
- *MIM* - "motion interface module". See section 2.1.2.
- *NC* - "numerically controlled".
- *NGIS, NGIS II* - "Next Generation Inspection System" is a cooperative research project funded by its members and by the National Center for Manufacturing Sciences (NCMS).
- *OLE* – “object linking and embedding”. “Microsoft’s object-based technology for sharing information and services across process and machine boundaries”. [6]
- *overrun* - the failure of the SIM to execute operations as fast as they are commanded. The typical operation commanded is to sample, convert, and store data. Overrun can

occur when the SIM's conversion time for all programmed channels exceeds the time between successive commands.

- *probe* - "a device that establishes location of the movable components of a coordinate measuring machine relative to a measurement point." [3] Types of probes that are in the scope of this NGIS specification are: displacement-measuring probe, proportional probe, proximity probe, and switching probe.

- *probe approach distance* - "the distance of approach to the part at which the machine traverse speed is reduced to the probe approach rate for measurement". [3]

- *probe approach rate* - "the nominal speed of approach of the probe toward the part during the acquisition of data (used primarily for switching probes)". [3]

- *proportional probe* - "a probe that gives a signal proportional to a distance between a reference point on the machine ram and the workpiece. Such probes may be displacement-measuring probes, [or] proximity probes..." [3]

- *proximity probe* - "a probe that gives a signal proportional to a distance from the probe tip to the workpiece". [3]

- *ram* - "the moving component of a CMM that carries the probe" [3]

- *sample* - during analog-to-digital conversion, the input of a signal to conversion circuitry.

- *sensor* - a device that responds to a physical stimulus and transmits a resulting signal. [4]

- *SIM* - "sensor interface module", a hardware card that plugs into the ISA computer bus and supplies an interface between a sensor and a machine control system.

- *SIM configuration file* - a file describing operating parameters of a SIM for a particular measurement task; called also ".ini file". It is typically generated by an operator using a software utility. Currently .ini files may be in any vendor specified format.

- *SIM Specification Sheet* - a file describing the capabilities of a SIM as supported by the NGIS SIM API, including hardware configuration, performance characteristics, and event programmability.

- *switching probe* - "a probe that gives a binary signal as a result of making contact with, or being in proximity to, a workpiece". [3]

- *touch-trigger probe* - a probe that produces an electrical pulse when it physically touches a part.

1.4 References

[1] *IEEE 1451.2 Standard for a Smart Transducer Interface for Sensors and Actuators – Transducer to Microprocessor Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats*, 1997. This effort addresses some of the same interface issues as

this document. A web site is located at URL http://129.6.36.211/Home/P1451/IeeeSite/P1451.htm.

[2] *IEEE Guide to Software Requirements Specifications*, ANSI/IEEE Std 830-1984.

[3] *Methods for Performance Evaluation of Coordinate Measuring Machines, ASME B89.4.1-1997.* Note: these definitions are quoted exactly from the standard. Some probes produce signals that are not mathematically *proportional* to the distance to the workpiece - they may yield nonlinear signals that must be calibrated to give distance.

[4] IEEE Std-100-1992, *New IEEE Standard Dictionary of Electrical and Electronic Terms.*

[5] NCMS web site, http://www.ncms.org. A site for the NGIS project is under the listing "Collaborative Project Portfolio".

[6] Microsoft Visual C++ V5.0 online help, Glossary.

1.5 Document Overview

This document contains six parts: I. SIM Requirements Specification, II. SIM Object Model, III. SIM Specification Sheet, Appendix A. SIM Conformance Testing Specification, Appendix B. SIM Communications Technologies, and Appendix C. SIM DLL API Specification. The Requirements Specification is a description of the NGIS II concept of a SIM and of its interactions with a control system. The Object Model is a logical description of a SIM: its objects, their attributes, and methods used to access the objects. These first two parts are intended to be independent of the actual communications techniques or platform configuration used (e.g. operating system, programming language). The NGIS SIM Specification Sheet allows SIM vendors to specify the hardware configuration, performance characteristics, and event programmability of a commercial product. The Conformance Testing Specification is a suite of SIM methods to be used to test compliance of any commercial product to the NGIS II SIM Specification. Appendix B is a discussion of potential communications technologies that may be suitable for implementing the SIM API. Appendix C describes the DLL implementation of the API.

2. General Description of the SIM

This section describes an overview of NGIS probes and NGIS controller models. Detailed description is in section II.3 and in *III. NGIS SIM Object Model.*

2.1 SIM Perspective

This section describes the SIM and its relationship to other components in a machine controller.

2.1.1 Industrial Application

This specification applies to commercial technology for inspection of manufactured parts. An example of its use involves the coordinate measuring machines in an automotive manufacturing facility. Parts inspected include engine and transmission components. Large manufacturers own several CMMs that may be purchased from different vendors. The probes included in NGIS II have all been developed in the past few years, to improve upon the functionality of touch-trigger probes. Manufacturers would like to use NGIS II probes and future technology on their CMMs without having to change controllers or to incur costly integration projects by each controller supplier. The ideal scenario that this specification may make possible is this: when a new probe is developed for CMM controllers that is NGIS II compliant, the manufacturer will buy one and integrate it with the CMM controller hardware and software on the shop floor, and begin productive operation immediately.

Some NC machine tools are capable of manipulating probes to do part inspection. The above scenario will apply to the addition of probes to NC controllers.

The concept of a SIM was developed to modularize control system functionality as a target for standardization. The goals of SIM standardization are:

- allow newly developed probes to be easily integrated into existing systems,
- complement machining with on-machine inspection for machine tool applications,
- support reduction of part variation by detection and correction of errors,
- have no negative effect on process throughput or system performance
- support inspection capability where part volumes are low,
- support data requirement of data analysis modules, e.g., recording a cloud of measured surface points,
- support requirements of computer aided design (CAD) modeling processes.

2.1.2 Machine Controller - block diagram

Figure II-2 NGIS Machine Controller Model, is a high level view of a CMM or NC machine tool control application.

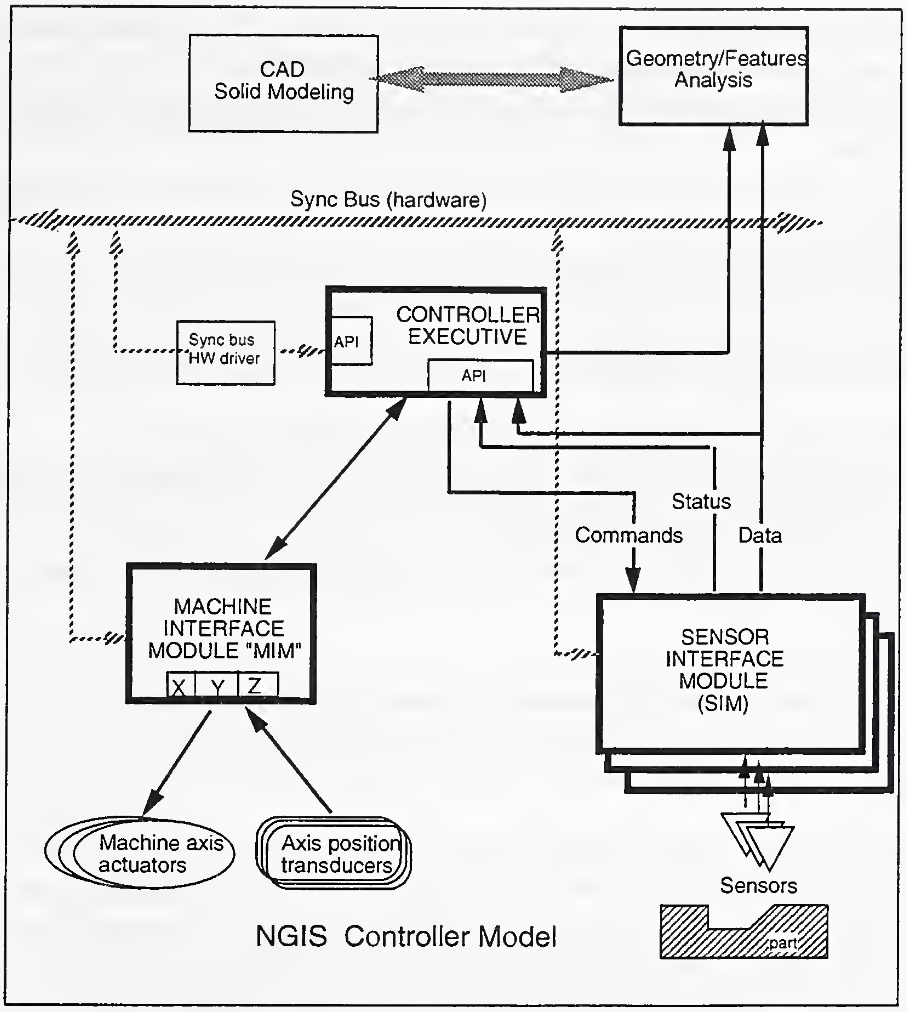


Figure II-2. NGIS Machine Controller Model

Controller Executive (CE)

- initializes the MIM and SIM for inspection tasks.
- coordinates the MIM and SIM using commands to each to accomplish inspection tasks.
- handles two-way file accesses, including reading inspection programs and outputting inspection data.
- processes input commands either from an automated supervisory process or from an inspection program (e.g. RS-274 or DMIS) and directs system actions.

Motion Interface Module (MIM)

- takes motion commands as input and generates velocity commands to machine actuators. The MIM performs real-time calculations using machine geometry to cause

motions such as straight line motion and circular arcs. During inspection, the MIM uses axis position data to calculate the probe tip location.
- reads machine axis position sensors and makes the data available to other parts of the controller.
- can generate a sync pulse output to signal other components of the controller to capture data.
- can interpret a sync bus event as a signal to perform an action.

Sensor Interface Module (SIM)

- responds to external commands for sensor sampling or run-time .
- samples and digitizes sensor signals and makes the data available to the controller .
- calculates engineering units (EU).
- can generate a hardware signal output (sync bus) on certain conditions of the sensor signal.
- reports its status.
- can interpret a sync bus event as a signal to perform an action.

Sync Bus

The sync bus is a two-way hardware bus that conveys voltage levels and transitions to signal components to perform actions or to signal that a particular condition has been reached. It is used primarily for very fast communications to synchronize actions of different components. Sync bus use includes causing simultaneous sampling of machine axis data and probe data, and signaling of probe thresholds reached (e.g., for touch trigger emulation and signaling a probe crash condition).

Geometry and Features Analysis

- uses arrays of data about the probe tip location or probe tip contact point during inspection to calculate fits of geometric features.
- determines differences between the calculated fits and desired part geometry to yield part error. The part error is compared to the allowable part tolerances.

2.2 SIM Functions

2.2.1 General

SIMs sample electrical signals from sensors and provide digitized analog and binary data to motion controllers in real time. SIMs also can receive information from the controller about SIM configuration to accommodate different operating modes.

Probes targeted by this document include proximity, one-dimensional feature measurement, and surface finish. SIM-based probes measure and report analog quantities. They also have dedicated processors that can support signal filtering and conditioning, conversion to engineering units, touch-trigger emulation, and crash detection.

2.2.2 Part inspection

SIMs can support two types of probing: single-point measurement and scanning. In single-point probing the SIM produces a simulated "touch" signal when an analog probe signal threshold is reached during slow-speed probe motion toward the part. Scanning inspection involves recording analog probe values while the probe is moved along or near the surface of a part.

At any time, there may be several SIMs plugged into the machine control computer backplane. Statements in the CMM inspection program select the SIM and probe for a task. Before probe motion begins, the controller establishes communications with the specific SIM and configures it for the inspection task.

2.2.2.1 Single Point Measurements

Since current NGIS probes are analog, the SIM must be configured for touch-trigger emulation by the CE. The SIM action is a sync pulse or level output to the MIM when a probe signal threshold is detected. The MIM stops motion and records the value of the machine axis positions.

2.2.2.2 Scanning Inspection

Scanning involves moving a probe across the surface of a part so that the surface is always within the probe's sensing range. Measurements of probe tip position are made by recording simultaneous readings of the arm position and probe data during real-time control cycles of the machine controller. Measurements may be recorded according to time frequency (Hz) or according to distance traveled by the probe tip (e.g., a measurement is made every .1 mm). A CMM may have several different probes available.

The machine controller begins a part scan by moving the probe so that the part is in the probe's sensing range, with a touch-trigger operation. The machine controller reads the probe data in real-time, and stops machine motion when the data reaches a programmed threshold. Some SIMs can be programmed to recognize the threshold and then to issue a sync pulse to signal machine motion to be stopped. Other SIMs report continuous proximity data and the machine controller must poll for the threshold.

Scanning is begun, directed by motion statements in the CMM program. For a servoed scan, the controller must read probe data and change the motion of the machine to keep the probe from getting too close (crash) or too far away from the part. Probe tip locations are calculated by recording simultaneous values of machine axes positions and probe readings.

For polled and scanning measurements, the SIM must produce readings at rates between 100 and 1000 Hz.

2.3 User Characteristics

SIM users are control system suppliers and manufacturers who own CMMs and NC maching tools. Probes with their SIMs will be purchased as off-the-shelf products. The SIM hardware and software cannot be changed by users to make it compatible with a controller. Thus this specification must provide for vendor-independent compatibility of SIMs and the controller platforms in which they are installed.

2.4 General Constraints

Measurement capabilities of probes must be matched to controllers to ensure the usefulness of probe data and to ensure the safe operation of the CMM or machine tool. Controllers must "understand" the data supplied by the SIM.

2.5 Assumptions and Dependencies

SIM functionality currently covers inspection tasks with proximity probes only (e.g., excluding camera vision).

Computer platform requirements, including ISA bus, NT operating system, and DLL communications technology, were chosen to limit the complexity of the spec and to encourage product development using commercial off-the-shelf technology that is currently in extensive, relatively low-cost use.

3. Specific Requirements for SIMs

3.1 Functional Requirements

Figure II-3. NGIS SIM Functional Description shows a model of internal functionality of the SIM and the external data flow requirements for a SIM. Not all SIMs will implement all the internal functions shown or handle all the external data described. Vendor specification sheets will describe supported functions. For automated configuration of an executive's interactions with the SIMs, "SIM Specification" files will be needed to describe each SIM's capabilities. This document does not address the content or format of such files. Further, API calls that are not supported will return error status.

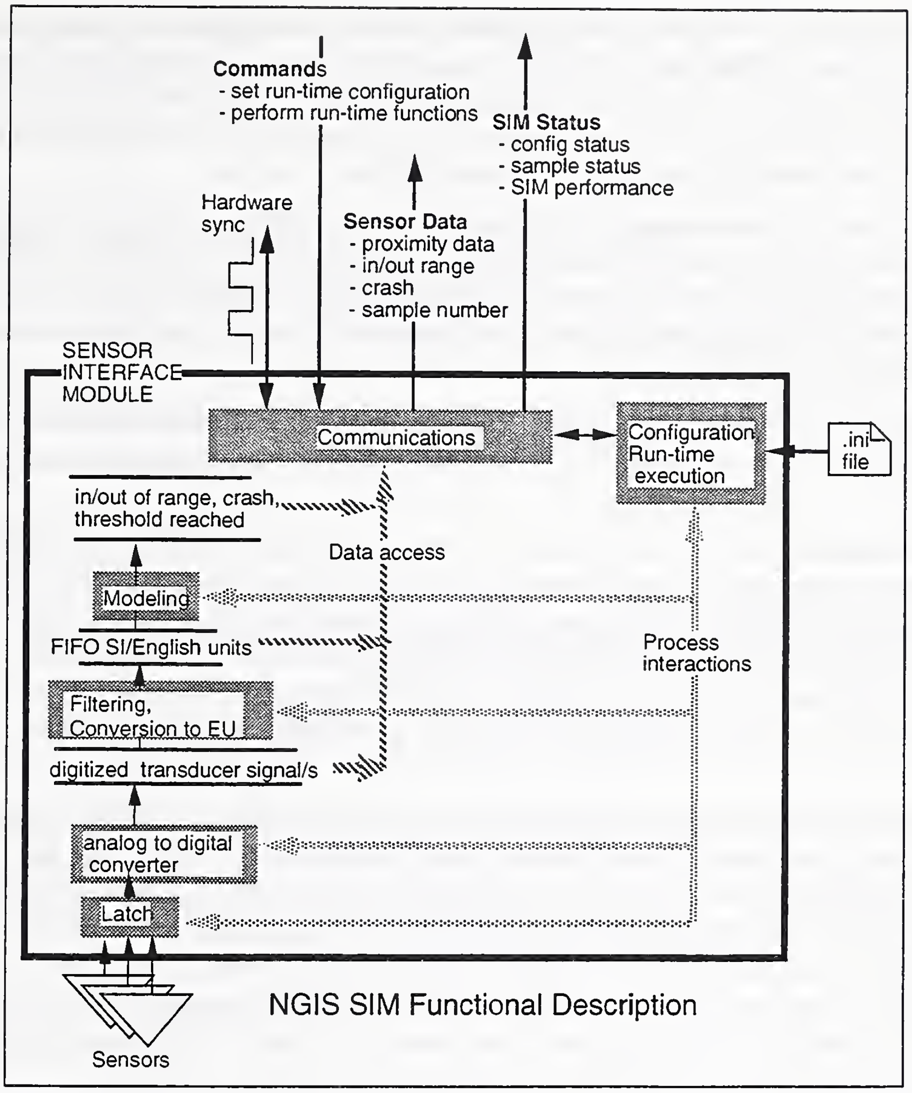


Figure II-3. NGIS SIM Functional Description.

3.1.1 Internal SIM Data and Functions

The internal functions and data shown in Figure II-3 are described below in bottom-up fashion. This is a generalized model and may not apply exactly to any specific commercial SIM. Not all SIMs will implement all items shown.

Sensors are electrically connected to the SIM using various technologies, which are not in the scope of this document.

The **latch** may have multiple channels for connections to several sensors. It has circuitry to sample electrical signals and present them to the analog-to-digital (a/d) converter. The latch has interfaces to it that tell it which amplifier gains to use, which channel(s) to sample, or a command to ‘take a sample now’.

The **analog-to-digital converter** (A/D) changes the sensor signal to digital data. The A/D conversion time is typically a significant part of a SIM's latency time. **Digitized sensor signals** may be "visible" outside the SIM, or they may be hidden.

The **filtering and conversion to engineering units** process applies calibration curves or equations to the digitized data and outputs engineering units in either English or metric units. The filtering may use several past samples plus the current one to generate an output. This function has inputs that tell it what kind of engineering units to calculate, what calibration parameters to use, what filtering algorithm to use, etc.

Modeling uses engineering units or digitized sensor signals to recognize conditions that will be useful to other controller components. Typical conditions are object in/out of probe range, probe "crashed" into the object, an engineering unit or probe signal threshold has been reached. Modeling has inputs that tell it whether a threshold should be checked, what threshold value to use, etc.

Not all SIMs will implement modeling. The task of recognizing the conditions described above may have to be performed by the controller executive based on EU data exported by the SIM.

Communications handles data input to the SIM and gathers data produced by SIM functions and conveys it to other controller components. Media used by NGIS SIMs includes: the ISA bus, including dual-port-ram and port-based input/output(i/o), RS-232, Sync Bus. SIM input data are commands and their parameters. SIM output data can include: probe data, probe data conditions, and SIM status, including status of commands. The sync bus can convey commands to the SIM as well as probe data conditions from the SIM.

Configuration and run-time execution is the overall manager of SIM operation. It conveys data to internal functions before inspection has begun to configure the SIM. It controls execution of the functions during inspection. For SIMs that have changeable configuration parameters (inputs to the functions), the parameters can be placed in a vendor-specific initialization file called an ".ini file". The format of this file is not included in the scope of this document. Some SIMs will be configured by an operator building an ".ini" file of parameters, and the controller executive issuing a "configure-using-file" command to the SIM. Other SIMs are configured by external commands.

3.1.2 External SIM Interactions

SIM interactions with the rest of the controller occur through commands to the SIM and reporting of probe data and SIM status by the SIM.

3.1.2.1 SIM Commands

SIM Commands cause real-time behavior or configure modal characteristics of the internal functions.

Run-time commands include:

- Sample probe signal on channel *x*, and save the analog data.
- Sample probe signal on channel *x*, convert it to EU and save the data.
- Sample and save on channel *x* at a rate of *y* Hz..
- Present the last sample value taken.

- Present the entire current FIFO of sample values.
- Clear the FIFO of values.

Configuration commands are:

- Reset the SIM.
- Configure SIM using parameters in named file (".ini file") .
- Perform a sample and save on sync bus event *e*.
- Generate a sync bus event *e* when probe data threshold is reached (touch-trigger emulation).
- Set probe data threshold value to x.
- Generate a sync bus event *e* when probe crashes.

3.1.2.2 External SIM Data

Probe data can include:

- probe deflection or distance from probe to object as raw sensor data or engineering units.
- object is in/out of range of probe.
- probe is in contact with object .

SIM Status can include:

- status of last sample.
- status of last configure command.

3.2 External Interface Requirements

3.2.1 Hardware Interfaces

Hardware interfaces to the SIM are limited to one or a combination of the choices below:

- ISA bus.
- dual port ram.
- port I/O.
- RS-232.
- Sync bus.

3.2.2 Software Interfaces

All software interfaces to a SIM must be through drivers that comply with the NGIS SIM API.

3.3 Performance Requirements

Real-time motion control requires high-speed probe data conversion and communications with the MIM. Requirements for probe data can range from 50 hz to 2000 hz. Maximum

SIM data rates are affected by the combination of times to perform analog-to-digital conversion, filtering, conversion to engineering units, and communication of data to the MIM or CE.

3.4 Design Constraints

This section addresses need to comply with any standards, and hardware constraints for SIMs. It will be expanded by the SIM Working Group.

3.5 Attributes

This specification describes an element of open architecture controller structure. That is, the interfaces to a SIM are specified and agreed to so that multiple-vendor products will be interoperable. However, the vendor's implementation of sensor and SIM hardware, and API software driver may remain proprietary.

3.6 Other Requirements

SIM Specification Sheet

This document does not deal with the issue of describing the capabilities of a SIM for automatic configuration by the controller executive. The working group has discussed the use of a SIM specification sheet but has not specified file content or syntax.

Not all SIM characteristics can be identified through the API. For example, differences between SIMs can include number of sensor channels, choice of channel to use for a particular sensor, ability to change latch amplifier gains, output of the sensor signal versus engineering units, ability to output a sync pulse, memory or port i/o location for communications.

Here is an example SIM specification file.

```
; --- SIM specification file ---
; semi colon is start of comment
[GENERAL_SPECS]
PRODUCT = ACME_CONTACT_PROBE
VERSION = 1.2

[A/D_SPECS]
NUMBER_OF_A/D_CHANNELS = 3
MAX_SAMPLE_RATE        = 1000 ; Hz
MAX_FILTER_CUTOFF      = 1000 ; Hz
MAX_GAIN               = 100

[PARALLEL_I/O_SPECS]
NUMBER_OF_IN_CHANNELS  = 8
NUMBER_OF_OUT_CHANNELS = 8

; Probe spec info may contain the following data:
[GENERAL]
PRODUCT     = ACME_3D_LVDT_PROBE
VERSION     = 1.2
PROBE_TYPE = LVDT ; LVDT | CAP | ?
```

```
[LVDT]
MAX_MM       = 10 ; MM
MIN_MM       = .1 ; MM
DEGREE_OF_FITTING_POLYNOMIAL = 1 ; 1 = ax + b

; --- end of SIM specification file ------
```

3.7 Scenario of SIM Use

The following scenario describes a session to add a SIM to a machine control system, configure the SIM behavior to support 1) single-point measurements and 2) scanning inspection, and then operate the SIM as a component of the machine control system to perform inspection. Some SIM configuration actions are common to both inspection modes and will be described first. These scenarios are simplified and do not use all defined SIM functions. There is a more detailed scenario in section *III. NGIS II Object Model*, and *Appendix A. Specification for SIM Conformance Testing.*

SIMs can be configured by the controller executive using SIM commands or by issuing the "Configure using file" command. The format of the file is beyond the scope of this document and may be vendor specific.

3.7.1 Machine Control System and SIM Setup

1. Add vendor supplied SIM driver software to machine controller configuration. More details of this action are described in Appendix A.
2. Controller executive reads SIM Specification file.
3. Controller executive establishes communications with SIM.

3.7.2 Single-point Measurements

Configure SIM

Commands to SIM:

1. Set probe threshold value to *x*.
2. Generate sync bus event *e* on probe threshold.
3. Generate sync bus event *f* when probe crashes.

Controller Operation for Inspection

The controller executive interprets statements in the inspection program and sends motion commands to MIM to move in paths perpendicular to surfaces to be probed. When the SIM senses proximity of a part and the configured threshold is reached it issues a sync pulse or transition. The MIM has been configured to stop machine motion and read machine position when it "sees" a sync bus event.

After motion has stopped, the next motion statement is interpreted, and the probe is moved away from the part.

3.7.3 Scanning Inspection

The application is to scan a part surface and measure points (e.g. every .1 mm along the motion probe tip path).

SIM Setup

Commands to SIM

1. Configure the SIM to set channel *y* amplifier gain to *x*.
2. Perform a sample and save to FIFO on sync bus input level change.
3. Generate a sync bus event when probe crashes.

Controller Operation for Inspection

1. Controller executive issues SIM command: Clear the FIFO.
2. MIM moves machine to position probe close to the part.
3. MIM monitors machine position and issues a sync event every .1mm. It also records machine position in a local FIFO.
4. SIM responds to each sync by sampling and storing probe data in its FIFO.
5. At the end of the scan the controller executive gathers the arrays of data from the MIM and SIM. The data is exported to the Geometry/Analysis process.

3.7.4 Error Detection

An error can occur during scanning measurements if the FIFOs of the SIM and MIM are not the same length at the end of the scan. This skewing of data may occur if an erroneous sync bus event was generated or if the SIM or MIM cannot sample the data fast enough to keep pace with sync bus commands. The error detection strategy is for the CE to compare the length of the two FIFOs and report an error if they are not equal.

III. NGIS SIM Object Model

1. Overview

The SIM Object model describes the internal data, internal functions, and external interactions introduced in section I.2. The SIM Object Model is intended to be independent of the choice of operating system, communications technology, and programming language used to implement the SIM or controller executive. This section describes the SIM object model using the computer language C++. Sections of C++ code are placed between horizontal lines as shown below.

```
C++ code goes here.
```

To accommodate a range of potential SIM capabilities, levels of SIM functionality will be defined. Level 1 defines capabilities that must be supported by all SIMs. Advanced SIM technology that may be used in the future will be defined in Level 2, Level 3, . . ., but vendor support is optional. This document addresses Level 1 only.

There are two main classes in the model: SIM Manager and SIM Task. See Figure III-1. SIM Manager comprises methods used by the controller executive that perform SETUP, CONFIGURATION, and TASK CONTROL. SIM Task methods are grouped into categories to perform DATA SETUP, OUTPUT, DATA RETRIEVAL, EVENT HANDLING, EXECUTION and MONITORING. These methods cause very specific actions to be performed by the SIM for probe data acquisition, conversion, and conveyance of control and status data to other controller components.

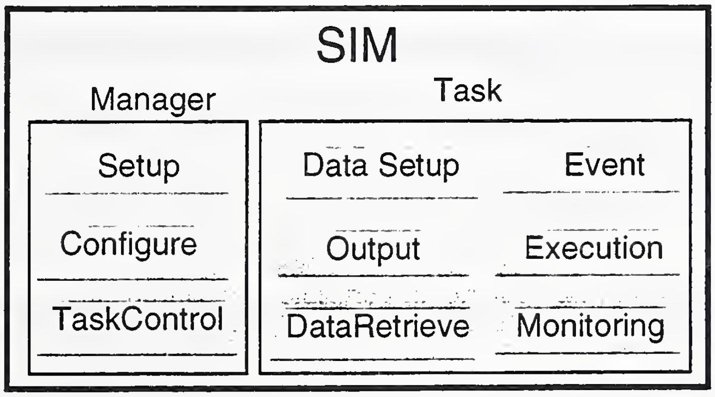


Figure III-1. The SIM Object Model

2. SIM Manager Model

This section describes the methods of the SIM Manager (SIMMgr) class. SIM management assumes that only one task exists at a time. No "blending" of SIM tasks can occur. It is further assumed that configuration and calibration are one-time activities that are done before a SIM task is "run." The SIM manager performs the following operations:

SETUP - establish communications with a SIM, load a SIM manager executable from a file, get status, get SIM-specific identification strings, restart the SIM manager, and query SIM error messages.

CONFIGURATION - establish modal attributes of the SIM that will affect probe signal reading, and data modeling and conversion functions.

TASK CONTROL - for the SIM Task object, perform creation, start, stop, restart, and deletion.

Each of the functional areas are for single communication between the SIM and the controller executive. The SIM manager does not run as a separate thread of execution typical of an executing agent. An example is a SIM with a dual port ram interface. On the ISA BUS, there is no way for a "slave" card to change memory or call routines on the main processor. In the current DLL-based communications, there is only a "pull model" from the main processor's perspective. The idea of the SIM manager executing in the DLL may be restated in our context as, the SIM manager routines are contained in the DLL.

2.1 Setup Methods

2.1.1 SIM Identification Methods

The Identification methods provide SIM-specific information to the application such as vendor, model, and version.

`char * ` **`SMgetManufacturersId`**`();`
  Returns pointer to a string, e.g., "Extrude Hone", "API", "SAMI".

`char * ` **`SMgetModelNumber`**`();`
  e.g., "AccuProbe 123"

`double ` **`SMgetRevisionCode`**`();`
  e.g., “V1.1”

`char * ` **`SMgetSerialNumber`**`();`
  e.g. "NIST 552885"

`char * ` **`SMgetDateCode`**`();`
  Returns release date - "mm/dd/yy-hr:min"

`char * ` **`SMgetProductDescription`**`();`
  Returns brief blurb to describe SIM connection.

`char * ` **`SMgetLicenseOwner`**`();`
  e.g. to whom product is licensed.

2.1.2 Strapping Method

The strapping method defines the communications medium and parameters so that API software can communicate with the SIM. Types of SIM strappings include:

- Dual port ram
- I/O bus
- Serial port
- Parallel port

```
enum  STRAP_TYPE {DUAL_PORT_RAM,
                  IO_BUS,
                  SERIAL_PORT,
                  PARALLEL_PORT };

class Strap             {
             public:    // Unclear if any base class is required for this.
            // long type; } ;
class dualPortRam:Strap  { public:
                            long  type;   /* = 1 */
                            ulong physaddr ; } ;
class IOBus:Strap        { public:
                            long  type;   /* = 2 */
                            ulong physaddr ; } ;
class SerialPort:Strap   { public:
                            long  type;    /* = 3*/
                            long  port; } ;
class ParallelPort:Strap { public:
                            long  type;   /* = 4 */
                            long  port ; };
```

SIMerror **SMstrap**(STRAP_TYPE strap, long address) ;
    Establish communications with the SIM using one of four media.

2.1.3 SIM Manager Startup Methods

SIM Manager startup methods cause transitions between the DOWN, RESET and READY states. The SIM Manager methods are implemented in software that implements the API and possibly coupled with software physically running on the SIM. Software that runs on the SIM is the "SIM Executive". The executive may be resident as firmware, or may need to be downloaded into SIM program memory. Not all SIM implementations will have a SIM executive, but all SIMs have the concept of a SIM Manager.

```
enum RESTART_TYPE { COLD, HOT } ;
```

`SimError` **`SMRestart`**`(RESTART_TYPE start) ;`

Restarts the state of the SIM Manager. There are two restart types: COLD and HOT. This method clears any SIM state information established by the Configuration methods.

`ulong` **`SMAlive`**`() ;`

Used to query the SIM Manager to see if it is running. The number returned is a "heartbeat" count. There are no constraints on the frequency of the heartbeat, however it should be fast enough to be useful to the Controller Executive in assessing SIM Manager status in real-time. If there is a SIM executive, this method should reflect its running status.

```
SimError SMLoad(char * fullpathname) ;
```

Load and run the SIM Manager executable code (not a job) in the named file. Depending on the implementation, the load could be of code linked to the API, or it could result in a download to program memory on the SIM.

```
char * SMName() ;
```

Queries the name of the SIM manager. Probe model name is typically used here.

```
char * SMErrorMsg(SimError errnum) ;
```

Used to query for the ASCII text of a specified error number. The integer errnum is used for error identification. Two errnums, 0 and 1 are standard, others may be assigned by the SIM vendor. There are no standard error message strings.

2.2 SIM Configuration Methods

Configuration is the setting of parameters for a/d conversion, signal filtering, conversion to engineering units, and modeling processes, as shown in Figure II-3. Examples of parameters are operational amplifier gains, parameters of filtering algorithms, and coefficients of calibration equations that convert analog sensor signals to engineering units.

The use of an ".ini" file to set configuration parameters is optional, but the SMConfig method is currently the only standard way to convey parameters to the SIM. ".ini" file content and format is not addressed by this specification.

```
SimError    SMgetChannelInfo(       CHANNELS        &raw,
                                    CHANNELS        &processed,
                                    BINARY_CHANNELS &mft) ;
```

This method returns bitmasks of the channel capabilities of the SIM.

```
SimError    SMconfig(char * inputFileName) ;
```

This method tells the SIM to configure itself using information in a vendor-supplied file.

2.3 SIM Manager Task Control

The SIM Manager creates and controls the SIM Task. Only one SIM Task can be run at a time. Once a Task is started its primary function is to continuously monitor triggers and capture sensor data. Methods include the following:

```
typedef long  simId ;           // A handle for a SIM Task.
```

```
simId SMCreateSimTask ( void );
```

Creates the SIM Task. SimId is a unique identifier, or handle. Return of zero indicates failure.

```
SIMerror SMStartSimTask ( simId id);
```

The SIM Task must be started for it to monitor triggers, and capture and process sensor data. The FIFO is empty on start.

```
SIMerror SMStopSimTask ( simId id);
```

Stops the cyclic processing of the SIM Task. The FIFO is preserved.

```
SIMerror SMRestartSimTask ( simId id);
```

Resumes the cyclic processing of the SIM Task, using previous programming and FIFO.

```
SIMerror SMDeleteSimTask ( simId id);
```

Deletes the instance of the SIM Task: the handle will not exist.

2.4 SIM Manager Level 2

The Working Group has begun discussions on Level 2. The Level 2 SIM Manager Class will inherit all the functionality of Level 1 but provides the additional capability:

- Job Control - single stepping, pausing, resuming, etc.
- Multiple Jobs - perform SIM Task list management so that several jobs can be run consecutively without interruption.

2.5 SIM Manager C++ Class Definition

This section defines the Sim Manager class using the C++ language.

```
class SIMMgrLevel1
{
public:
  // Constructor for binding to software driver module.
  SIMMgrLevel1(char * driverFileName) ;    // e.g. a DLL

// IDENTIFICATION
char * SMgetManufacturersId();
char * SMgetModelNumber();
double SMgetRevisionCode();
char * SMgetSerialNumber();
char * SMgetDateCode();
char * SMgetProductDescription();
char * SMgetLicenseOwner();

// STRAPPING
enum STRAP_TYPE {DUAL_PORT_RAM, IO_BUS, SERIAL_PPORT, PARALLEL_PORT } ;

class Strap { public:
             // Unclear if a base class is required for this.
             // long type;
             };
class DualPortRam:Strap    { public:
                              long  type; /*= 1 */
                              ulong physaddr; };
class IOBus:Strap          { public:
                              long  type; /* = 2 */
                              ulong physaddr ; };
class SerialPort:Strap    { public:
                              long type; /* = 3*/
                              long port; };
class  ParallelPort:Strap { public:
                              long type; /* = 4 */
                              long port ; };

SIMError      SMstrap(STRAP_TYPE strap, long address) ;

// RESTART
enum RESTART_TYPE { COLD, HOT } ;

SimError  SMRestart(RESTART_TYPE start) ;
ulong     SMAlive() ;
SimError  SMLoad(char * fullPathFileName) ;
char *    SMName() ;
SimError  SMErrorMsg(SimErrror errNum) ;

// CONFIGURATION
SimError  SMConfig(char * configFileName) ;
SimError  SMgetChannelInfo(CHANNELS &raw,
                           CHANNELS &processed,
                           BINARY_CHANNELS &mft) ;

// TASK CONTROL
typedef long simId ;
simId     SMCreateSimTask  ( void );
SIMerror  SMStartSimTask   ( simId id);
SIMerror  SMStopSimTask    ( simId id);
SIMerror  SMRestartSimTask ( simId id);
SIMerror  SMDeleteSimTask  ( simId id);
}
```

3. SIM Task Model

The SIM task has seven categories of methods. A prefix is associated with each category.

SimTask Method Categories	Prefix
Configuration	CFG
DataSetup	DC
Event	EV
Execution	EX
Output	OUT
Monitor	MON
Data Retrieval	DR

There are no SIM Task *Configuration* methods. The *Data Setup* methods specify channels to sample and process. The *Event* methods define standard triggers for data capture. The *Execution* methods assign a number of SIM Task executions to the trigger. The *Output* methods communicate with SIM I/O hardware. The *Monitor* methods query the status of an executing SIM Task. The *Data Retrieval* methods retrieve stored data records and give information about the FIFO.

3.1 The Data FIFO

The FIFO is a shared buffer used to convey data from the SIM Task to the user. The SIM Task stores "raw" values as signed 32-bit integers: they represent the output of a/d conversion of a channel signal. "Processed" values are stored as IEEE 32-bit signed floating point numbers and represent engineering units. "Manufacturer's" values are stored as a 32-bit word of binary encoded data. The order of the data in the FIFO record is all raw data in channel order, lowest channel first, all processed data in channel order, then manufacturer's data.

A FIFO record is all of the data produced by the SIM Task when it responds to one trigger. For example, if three channels were programmed, each FIFO entry will contain three 32-bit words of information. If the maximum memory allocated for the FIFO is used, new data values are lost. Existing FIFO entries will not be overwritten.

3.2 Data Setup Methods

Data setup methods define which input channels will be read and what processing will be done on the signals to produce FIFO records. The SIM Task responds to a trigger by capturing data from all designated channels, performing processing, and writing a FIFO record. All three types of data can be mixed in a FIFO record. Some SIMs may support raw or processed data but not both.

A SIM can have up to 32 channels of analog input, and 32 binary inputs. A channel is a single signal input to the SIM that can convey either an analog or a binary signals. Individual channels and associated processing are selected by setting bits in three 32-bit masks, the raw mask, the processed mask, and the manufacturer's data mask.

The following mask is defined to select analog input channels:

```
typedef struct CHANNELS              // This is bit field definition. Set a
{                                    // bit to 1 to select the channel.
  unsigned    Channel0   :1;         //
  unsigned    Channel1   :1;
  unsigned    Channel2   :1;
  unsigned    Channel3   :1;
  unsigned    Channel4   :1;
  unsigned    Channel5   :1;
  unsigned    Channel6   :1;
  unsigned    Channel7   :1;
  unsigned    Channel8   :1;
  unsigned    Channel9   :1;
  unsigned    Channel10  :1;
  unsigned    Channel11  :1;
  unsigned    Channel12  :1;
  unsigned    Channel13  :1;
  unsigned    Channel14  :1;
  unsigned    Channel15  :1;
  unsigned    Channel16  :1;
  unsigned    Channel17  :1;
  unsigned    Channel18  :1;
  unsigned    Channel19  :1;
  unsigned    Channel20  :1;
  unsigned    Channel21  :1;
  unsigned    Channel22  :1;
  unsigned    Channel23  :1;
  unsigned    Channel24  :1;
  unsigned    Channel25  :1;
  unsigned    Channel26  :1;
  unsigned    Channel27  :1;
  unsigned    Channel28  :1;
  unsigned    Channel29  :1;
  unsigned    Channel30  :1;
  unsigned    Channel31  :1;            // most significant bit (MSB)

}  CHANNELS;                            // 32 Bits


This mask is used to select binary channels:

typedef struct BINARY_CHANNELS          // This is bit field definition.
{                                       // Set a bit to 1 to select the
        unsigned    SyncF       :1;     // channel.
        unsigned    Sync0       :1;
        unsigned    Sync1       :1;
        unsigned    Sync2       :1;
        unsigned    Sync3       :1;
        unsigned    Sync4       :1;
        unsigned    Sync5       :1;
        unsigned    Sync6       :1;
        unsigned    ManBit0     :1;
        unsigned    ManBit1     :1;
        unsigned    ManBit2     :1;
        unsigned    ManBit3     :1;
        unsigned    ManBit4     :1;
        unsigned    ManBit5     :1;
        unsigned    ManBit6     :1;
        unsigned    ManBit7     :1;
        unsigned    ManBit8     :1;
        unsigned    ManBit9     :1;
        unsigned    ManBit10    :1;
        unsigned    ManBit11    :1;
        unsigned    ManBit12    :1;
        unsigned    ManBit13    :1;
        unsigned    ManBit14    :1;
        unsigned    ManBit15    :1;
        unsigned    ManBit16    :1;
        unsigned    ManBit17    :1;
        unsigned    ManBit18    :1;
        unsigned    ManBit19    :1;
        unsigned    ManBit20    :1;
        unsigned    ManBit21    :1;
        unsigned    ManBit22    :1;
        unsigned    ManBit23    :1;      // MSB

}  BINARY_CHANNELS;                      // 32 Bits
```

Correspondence of channel numbers to sensor axes is defined by the manufacturer in the SIM specification sheet.

```
SIMerror  DCsetup(CHANNELS         raw,
                  CHANNELS         processed,
                  BINARY_CHANNELS  manufacturer_data
                  );
```

This method tells the SIM Task which channels to read and the type of processing to perform. All three arguments are 32-bit bit-masks. "raw" channel processing will be basic a/d conversion. "processed" channel signals will be converted to engineering units. "manufacturer_data" designates binary channels whose data will be stored as bits. manufacturer_data channels include six sync bus channels. The implementation of this method should report an error if the commanded type of processing is not supported or if the selected channel does not exist.

Example: Program the capture of x and z axis raw, y axis processed, and the syncF channel. The sensor's x-axis is the least significant channel, channel0.

```
    SIMerror    errType ;
                  //axis       zx       y       s-syncF
    errType  =    DCsetup( 0x0005, 0x0002, 0x0001);


  The FIFO after one trigger is 16 bytes long and looks like:

                  ---------------------------------
                 | xraw  |  zraw  |  yproc |  manu |
                  ---------------------------------

          Byte  0       4        8         12        16
```

```
int DCqueryFIFORecordLength() ;
```

This method returns the number of bytes in one FIFO record. It is used after a DCsetup method to verify configuration.

```
SIMerror DCallocateFifoSize(int num_bytes) ;
```

Command the SIM Task to allocate a total FIFO memory of "num_bytes" 32-bit words.

```
int DCgetMaxFifoSize(void) ;
```

Query the maximum amount of memory allocated to the FIFO. Units are bytes.

3.3 Defining Triggers

These methods use a standard set of events to define SIM triggers. A SIM responds to triggers by capturing sensor data to its FIFO. An output method can command the SIM to also generate an output signal such as a sync bus event. Types of triggers are: sync bus events, elapsed time, signal thresholds reached, and software commanded trigger. Once a trigger has been defined the SIM Task monitors for the event condition continuously.
There can be only one sync bus or elapsed time trigger defined at any time. However, separate signal threshold event triggers can be defined for each individual analog input channel. When any of the threshold triggers occurs, all programmed data channels will be read and processed. Programmed triggers are cancelled only by deleting the SIM Task (SMDeleteSimTask) and starting a new SIM Task (SMStartSimTask).

```
// Sync bus events - level transitions
typedef enum
 {
        TRANS_0         = 1,     // level is 0
        TRANS_1         = 2,     // level is 1
        TRANS_RISE      = 3,     // level changes from 0 to 1
        TRANS_FALL      = 4,     // level changes from 1 to 0
        NO_EVENT        = 5      // Cancel any previous trigger.

}  TRANS_TYPE;

// Events for sensor data channels.

typedef enum
{
        GREATER_THAN     = 1,
        LESS_THAN        = 2,
        GREATER_EQUAL    = 3,    // ie greater than or equal
        LESS_EQUAL       = 4,    // ie less    than or equal
        EQUAL            = 5

}  COMPARE_TYPE;


SIMError EVsetTriggerStore(BINARY_CHANNELS SyncChannels,
                           TRANS_TYPE      transition);
```

Define a sync bus event as a trigger. The sync bus event is an input to the SIM. Only one sync bus event may be programmed as a trigger. To cancel a sync bus trigger, stop the SIM Task, and then start it (SMStopSimTask, then SMStartSimTask).

Example: Define a trigger - the transition of the Sync1 channel from 0 to 1.

```
static BINARY_CHANNELS sync_channels; // Init to clear
SIMerror errType ;

sync_channels.Sync1 = 1;
errType = EVsetTriggerStore( sync_channels, TRANS_RISE );
```

```
SIMError   EVsetCounterStore(ulong timer_num,
                             ulong count_down_ms) ;
```

Define a periodic elapsed time event as a trigger. Units are milliseconds. The counter begins when the SIM Task enters the RUNNING state.

Example: Define a trigger - when timer 2 counts down from 100 ms.

```
SIMError errType;

errType  = EVsetCounterStore( 2,  100 );
```

```
SIMError EVsetThresholdStoreRaw
                        (CHANNELS     channel,   // channel mask
                         COMPARE_TYPE comparison,
                         ulong        value,
                         ulong        reset);
```

Define a trigger as a raw sensor channel value crossing the "value" threshold. The trigger is reinitialized when the channel value crosses the "reset" level.

Example: Define a trigger - when channel 1 raw value goes above 500, and reset when it gets below 400.

```
static CHANNELS channel;   // Init all bits to clear (0)

SIMError errType;

channel.Channel1    = 1;


errType = EVsetTriggerStore(       channel,
                                   GREATER_THAN,
                                   500,
                                   400 );
```

```
SIMError EVsetThresholdStoreProcessed
                        (CHANNELS     channel,
                                       COMPARE_TYPE comparison,
                                       float        value,
                                       float        reset) ;
```

Define a trigger as a processed sensor signal channel value crossing a threshold. The trigger is reinitialized when the channel value crosses the "reset" level.

Example: Define a trigger when channel 1 processed value goes below 23.1, and reset when it gets above 25.4

```
    static CHANNELS  channel;    // Init all bits to clear (0)
    SIMError errType;
    channel.Channel1 = 1;

    errType = EVsetThresholdStoreProcessed(channel,
                                           LESS_THAN,
                                           23.1,
                                           25.4 );
```

```
SIMError EVcommandedStore( void )  ;
```

This method is a command that generates a SIM trigger immediately.

Example: the controller executive gathers a new reading of the sensor data.

```
    SIMError errType;

    errType = EVcommandedStore();
```

3.4 SIM Task Output Methods

These methods define SIM actions to manipulate binary output channels. Actions can be to zero, set, or toggle an output level. Typically, outputs are used to notify a motion board or other slaved piece of hardware that a SIM trigger has occurred.

```
SIMError OUTsetOutput(BINARY_CHANNELS zero_channels,
                      BINARY_CHANNELS one_channels,
                      BINARY_CHANNELS toggle_channels) ;
```

Write to selected binary output channels when a trigger occurs. If the trigger was defined with a "reset" value, when the reset condition occurs all channels changed by the trigger will be inverted. (e.g. a zero_channel set low on a trigger, will be set high on the reset condition).

Example: configure the SIM to respond to a trigger by generating a rising transition on sync channel 2, and toggling the Man16 channel when a trigger occurs.

```
BINARY_CHANNELS   zero_ch   = 0;
BINARY_CHANNELS   one_ch    = 0;
BINARY_CHANNELS   toggle_ch = 0;
SIMError errType;

zero_chan.Sync2    = ENABLE;
toggle_ch.ManBit16 = ENABLE;

errType = OUTsetOutput(zero_ch, one_ch, toggle_ch);
```

3.5 SIM Task Execution

These methods specify data capture behavior while the SIM Task is running.

```
SIMError EXsetNumberExec(int count);
```

This method specifies the maximum number of data captures to perform while the SIM Task is running. The SIM will not respond to triggers after "count" triggers have occurred, until the SIM Task has been stopped and started again.

Example: Configure the SIM to respond to a maximum of 100 triggers.

```
SIMError errType;

errType = EXsetNumberExec(100);
```

```
SIMError EXContinuous( void );
```

Once the SIM Task is running, respond to triggers until the FIFO is full. Note: there is no explicit status of a full or overrun FIFO.

3.6 SIM Task Monitor Methods

These methods query the SIM task concerning its data and status.

```
// Status values of the SIM Task.

typedef enum
{ DONE            = 1,    // Finished in an orderly fashion.
  EXEC            = 2,    // Executing without error.
  FAIL            = 3,    // Failed, must be restarted.
  PAUSE            = 4,    // Active but paused.
  ABORT            = 5,    // Aborted.  Must be recreated.
  IDLE             = 6     //
  }  STATUS;
```

```
STATUS MONgetStatus(void) ;
```

This method returns status of the SIM Task.

```
SIMError  MONgetData(CHANNELS         raw_channels,
                     CHANNELS         processed_channels,
                     BINARY_CHANNELS manufacturers_channels,
                     void * loc,
                     int    record_size_match_check) ;
```

This method commands the SIM Task to capture and process channel data and write one record to the user's address "loc". MONgetData does not cause a FIFO entry or change any entries in the FIFO. "record_size_match_check" is the assumed size of the record in 32-bit words. If the assumed size is not correct, this method will return a non-zero (not SUCCESS) SIMError.

```
typedef enum
 { NO_ERROR         = 0,
   ESTOP            = 1,
   ABORTED          = 2,
   NOT_SUPPORTED    = 3
 }  ERROR_TYPE;
```

```
ERROR_TYPE MONgetError(void) ;
```

Returns a error value if the Task has aborted.

3.6 SIM Data Retrieval

These methods retrieve data records from the FIFO. Retrieval is of complete fixed-size records.

```
SIMerror DRretrieveRecords(
                 int  num,       // number of records.
                 void *loc,      // pointer into user space.
                 int  record_size_match_check); // bytes.
```

Removes "num" records from the FIFO and moves them to user space at "loc". Records are removed from the "bottom" of the FIFO: the most recent entries are on the "top". The number of stored records in the FIFO is updated to reflect this retrieval.

```
int  DRgetNumRecords(void);
```

Returns the number of records currently in the FIFO.

3.7 SIM Task C++ Class Definition

```
typedef enum
 {
        TRANS_0         = 1,
        TRANS_1         = 2,
        TRANS_RISE      = 3,
        TRANS_FALL      = 4,
        NO_EVENT        = 5
} TRANS_TYPE;

typedef enum
{
        GREATER_THAN      = 1,
        LESS_THAN         = 2,
        GREATER_EQUAL     = 3,
        LESS_EQUAL        = 4,
        EQUAL             = 5
}  COMPARE_TYPE;

typedef enum
 { NO_ERROR         = 0,
   ESTOP            = 1,
   ABORTED          = 2,
   NOT_SUPPORTED    = 3
 }  ERROR_TYPE;

typedef enum
{ DONE              = 1,    // Finished in an orderly fashion.
  EXEC              = 2,    // Executing without error.
  FAIL              = 3,    // Failed, must be restarted.
  PAUSE             = 4,    // Active but paused.
  ABORT             = 5,    // Aborted.  Must be recreated.
  IDLE              = 6     //
  }  STATUS;


class SIMTask
{
// SimTask Method Prefixes:
//  Config :         CFG
//  DataSetup :      DC
//  Event :          EV
//  Exec :           EX
//  Output :         OUT
//  Monitor :        MON
//  Data Retrieval : DR


public:
  // Miscellaneous functionality
  #define MAX_ERROR_MESSAGE_SIZE 256
```

```
// DATA SETUP

// Define channels to capture.
SIMError DCsetup(CHANNELS                raw,
                 CHANNELS                processed,
                 BINARY_CHANNELS         manufacturer_data
                 );

// Query FIFO record length - returns length of FIFO in bytes.
int       DCqueryFIFORecordLength();
// Query FIFO buffer space allocated - returns size in bytes.
int       DCgetMaxFifoSize(void);
// Reserve buffer space for the FIFO.
SIMError  DCallocateFIFORecordLength(int num_bytes) ;

// EVENT TRIGGER PROGRAMMING
// Define binary triggers.
SIMError EVsetTriggerStore(BINARY_CHANNELS   SyncChannels,
                           TRANS_TYPE        type );

// Define an elapsed time trigger - units are milliseconds
SIMError EVsetCounterStore(ulong timer_num,
                           ulong count_down_ms  );

// Define raw threshold triggers.
SIMError EVsetThresholdStoreRaw(CHANNELS        channel,
                                COMPARE_TYPE    comparison,
                                ulong           value,
                                ulong           reset);

// Define processed threshold triggers.
SIMError EVsetThresholdStoreProcessed(CHANNELS     channel,
                                      COMPARE_TYPE comparison,
                                      float value,
                                      float reset);

// Command an immediate software trigger.
SIMError EVcommandedStore( void );

// Define output signals to produce on a trigger.
SIMError OUTsetOutput(  BINARY_CHANNELS zero_channels,
                        BINARY_CHANNELS one_channels,
                        BINARY_CHANNELS toggle_channels);

// Set maximum number of trigger responses to execute.
SIMError EXsetNumberExec(int count);
// Respond to triggers until the FIFO is full.
SIMError EXContinuous(void );

// MONITOR
// Command a data capture and conveyance of resulting sensor data.
SIMError MONgetData(CHANNELS raw_channels,
                    CHANNELS processed_channels,
                    BINARY_CHANNELS manufacturers_channels,
                    void * loc,
                    int record_sizes_match_check);
```

```
// Query status of SIM Task
STATUS    MONgetStatus(void);
// Query specific SIM Task error type.
ERROR_TYPE MONgetError(void);
// Get ascii string corresponding to error type.
SIMerror MONgetErrorText(char * str, int error_num);
// Query if SIM Task has stopped.
BOOL      MONisDone();
// Query if SIM Task is executing
BOOL      MONisExecuting();
// Query if SIM Task has stopped with error.
BOOL      MONisFailed();
// Query if SIM Task can be started.
BOOL      MONisReady();

// DATA RETRIEVAL
// Transfer records from the FIFO - size match units are words.
SIMerror DRretrieveRecords(int num,
                           void * loc,
                           int record_size_match_check);
// Query the size of the FIFO
int      DRgetNumRecords(void);

};
```

4. SIM Functional Scenario

This section contains a short program in C++ to illustrate using SIM object model definitions, configure a SIM, start a SIM Task, and program it to read and process sensor data.

4.1 Program Overview

The main steps are:

- Declare objects, and create a SIM Manager bound to a driver module.
- "Restart" the SIM Manager.
- Establish communications with the SIM hardware.
- Load the SIM executive if necessary, start it running.
- Configure the SIM for specific sensors, including calibration.
- Create a SIM Task.
- Define the input channels to be read and processed.
- Confirm properties of the FIFO.
- Define a trigger .
- Program outputs if needed (e.g. sync bus evnets.
- Check SIM Manager status.

- Start a SIM Task.
- Check SIM Task status.
- SIM captures and stores data into its FIFO.
- Verify FIFO length and retrieve FIFO records.

4.2 SIM Declarations

Object declarations include:

- SIM Manager. One SIM Manager must be declared for each SIM in a system.
- Strap reference.
- SIM Task.
- Channel variables for data capture programming.

```
#include "ngisSIM.hh"     // Class defs for SIM Task & Manager

// Declare SIM Mgrs, bind to API driver modules, e.g. a dll.
  SIMMgrLevel1 * SIM1 = new SIMMgrLevel1("VENDOR1.DLL");
  SIMMgrLevel1 * SIM2 = new SIMMgrLevel1("VENDOR2.DLL");

  SerialPort * mystrap = new SerialPort();

// Declare pointer to SIM executive executable
  char *loadFilename = "C:\SIM\VENDOR1\sim.exe";

// Declare pointer to name of the .ini file
  char *configFilename = "C:\SIM\VENDOR1\sim.ini";

  SIMerror  errType ;
  char      errorMessageBuffer[256] ;
  int alive, countdown, err, num_records;

  // Declare SIM task reference
  simId   * Task1;     // level 1 declaration,
  SIMtask * Task2;     // level 2 object oriented declaration

  // DECLARE local FIFO data type and local storage pointer
  struct FIFO {
    long ch1_raw;
    long ch2_raw;
    long ch3_processes;
  };
```

```
void * localBuffer;           // Pointer to user data buffer.

// Declare channel structures
CHANNELS  ch;
CHANNELS  raw;
CHANNELS  processed;
BINARY_CHANNELS  sync_channels;
BINARY_CHANNELS  zero_channels;
BINARY_CHANNELS  one_channels;
BINARY_CHANNELS  toggle_channels;
```

4.3 SIM Manager Bootstrap

Startup steps include restarting the SIM Manager, establishing communications with the SIM hardware, loading the SIM executive if necessary, and configuring the SIM for specific sensors to be used.

```
// RESTART: Do a "cold" restart of SIM Manager
if(errType = SIM1->SMrestart(SIMMgrLevel1::COLD))
  {
  fprintf(stderr, "SIM %s not responding\n", SIM1->name());
  MONgetErrorText(&errorMessageBuffer, errType );
  fprintf(stderr, "Error message: %s", &errorMessageBuffer);
  exit(-1);
};

// STRAP physical information, return 0 = worked, -1 err
mystrap->type = SERIAL_PORT;
mystrap->port = 0x01;

if(errType = S1->SMstrap(mystrap))
  {
  fprintf(stderr, "SIM %s could not strap hardware\n", S1->name());
  MONgetErrorText(&errorMessageBuffer, errType );
  fprintf(stderr, "Error message: %s", &errorMessageBuffer);
  exit(-1);
  };
```

After the strap, the SIM executive is loaded and run. Some executives are resident on ROM and do not need to be loaded.

```
// Load the executive on the SIM from a file
  if(errType = SIM1->SMload(loadFilename))
    {
    fprintf(stderr, "Load of SIM %s executive file %s failed\n",
                        SIM1->name(), loadFilename);
    MONgetErrorText(&errorMessageBuffer, errType );
    fprintf(stderr, "Error message: %s", &errorMessageBuffer);
    exit (-1) ;
    }
```

The next step is to configure the SIM executive. Configuration can include specifying calibration parameters for specific sensors. All external configuration data comes from an .ini file.

```
  if(errType = SIM1->SMconfig(configFilename)))
    {
    fprintf(stderr, "Config of SIM %s executive file %s
                    failed\n", SIM1->name(), configFilename);
    MONgetErrorText(&errorMessageBuffer, errType );
    fprintf(stderr, "Error message: %s", &errorMessageBuffer);
    exit (-1) ;
    }
```

4.4 SIM Task Creation and Programming

The next step is to create the SIM Task. Tasks are referenced by an integer handle. If a 0 is returned the function failed.

```
Task1 = SIM1->SMcreateSimTask();
```

For level 2, a reference to an object will be returned.

```
Task2 = SIM1->SMcreateSimTask("Name");
```

Now program data capture and check the size of the FIFO for correctness.

```
errType = SIM1->DCsetup(0x003UL,   // capture raw data,ch 1,2
                        0x004UL,   // capture processed, ch 3
                        0L);        // no manufacturers data

// FIFO record size should be:
```

```
//  3 channels * 4  bytes/word = 12 bytes
  if(SIM1->DCqueryFIFORecordLength() != 12 )
  {
   fprintf(stderr, "Warning: SIM %s FIFO record size does not match
SETUP\n",
   S1->name());
  }
```

Next, a trigger is programmed. The example is a threshold store when channel 1's processed value goes below 23.1 and reset when it gets above 25.4.

```
ch.Channel1 = ENABLE;
errType =  SIM1->EvsetThresholdStoreProcessed
                          (ch,    // channel 1
                           LESS_THAN,   // transition type
                           23.1,        // threshold value
                           25.4);       // reset value
```

Triggers signal the SIM to capture data and perform outputs if programmed. This example's output is to set sync bus 2 to zero and toggle manufacturing channel bit 16.

```
zero_channel.Sync2       = ENABLE;
toggle_channel.ManBit16 = ENABLE;

if(errType = SIM1->OUTsetOutput(zero_channel, one_channel, toggle_channel))
  {
  fprintf(stderr, "Error: SIM %s unable to register output\n",
                  S1->name());
  exit(-1);
  }
```

4.5 SIM Task Execution and Data Access

Once a SIM Task has been programmed, it can be run. The first check is of the SIM Manager's status.

```
errType = SIM1->MONisReady()  ;
if(errType)
  {
   fprintf(stderr, "Error: SIM Manager not ready.\n");
   exit(-1);
  }
```

Then start the task.

```
if(errType = SIM1->SMstartSimTask(Task1))
  {
   fprintf(stderr, "Error: SIM Manager %s could not start \
                     task\n", S1->name());
  }
```

SIM Task status can be checked with the following methods.

```
  errType = SIM1->alive() ;
  if(errType = MONisFailed())
    {
      fprintf(stderr, "Error: SIM Task Failed\n") ;
      exit(-1);
    }

 if(errType = MONisDone())
   {
      fprintf(stderr, "SIM Task has completed\n") ;
      exit(-1);
   }

 if(errType = MONisExecuting()
   {
    fprintf(stderr, "SIM Task is not executing\n") ;
    exit(-1);
   }
```

If a measurement task is complete, the FIFO data can be accessed. The controller executive can query the number of records and then retrieve them.

```
  num_records = SIM1->DRgetNumRecords();
  localBuffer = malloc(sizeof(FIFO) * num_records);
  errType = SIM1->DRretrieveRecords(num_records, localBuffer,
                                          num_records * 12 );
```

IV. NGIS Sensor Specification Sheet

The purpose of the NGIS Sensor specification sheet is to allow SIM customers to specify the hardware configuration, performance characteristics, and event programmability of the SIM they wish to purchase.

1. Scope

The NGIS SIM specification sheet describes the hardware and software features available in a compliant SIM. The NGIS SIM hardware specification defines the platform and configuration information for the SIM hardware (at this time explicitly assuming a slave card). It also describes some performance metrics and physical characteristics.

The SIM software specification sheet lists capabilities supported by the SIM. It is not necessary for the SIM to support all NGIS API programming features. For this reason, the specification sheet user can select individual SIM event programming features. It is expected that the SIM will support all other SIM API functionality.

2. Schematic Model

Figure I-3. NGIS SIM Functional Description, showed the inputs, outputs, and control data flow requirements. Not all SIM modules must support every requirement. The specification sheet enumerates the capabilities an individual SIM supports.

3. Specification Sheet

FEATURE	SELECTION
Hardware Connection	Slave Card Network _______________ Other
Slave Communication Interface	Dual Port Memory I/O Port Serial Port Parallel Port
Number of channels	________
Worst Case Update Time	______ milliseconds
A/d resolution	_____ bits
Worst Case Channel Sampling Frequency	_____ hz

Synchronization Clock Support	SyncBus None Other _______
Max FIFO Size	______ 32-bit words
Platform CPU	Intel _____ Alpha PowerPC Other ________ CPU rate ______
Platform OS	Windows 3.1 Windows 95 Windows/NT Windows/CE UNIX: ______ LINUX Other ____________
Platform OS Extension	RT Windows NT POSIX compatibility Other _________
Event Programming	Time Position Threshold Other __________
Correction Utilities	Scaling Linearization Other _______

NGIS II SIM Specification

Appendix A. Specification for NGIS SIM Conformance Testing

1. Purpose of This Document

Primary audience:	probe developers, controller developers.
Secondary audience:	anyone (end user or developer) who will integrate an NGIS compliant SIM with an NGIS compliant controller.

This section describes procedures to test compliance of commercial products with this specification. The procedures can be run by probe developers, controller developers, control system integrators, or end users. A probe must support these procedures exactly, to be judged compliant with the NGIS II SIM Specification.

2. Scope

This version covers Level 1 SIM capabilities. The function calls conform to the DLL technology API specification defined in "sim.h".

In these sections the term *probe* is used to represent a SIM plus the sensor attached to it. The performance of a probe is a combination of the performance of the sensor and the computer hardware and software on the SIM.

This section does not cover considerations of sensor geometry or accuracy needed for specific inspection tasks or part geometry. It only addresses SIM hardware, and software interfaces to the SIM.

3. Measurement Tasks

This section describes measurement tasks that could be performed using an NGIS compliant probe. The task descriptions, plus specific values of parameters can be used in two ways: 1) describe required probe performance for an application -- this would be useful for a controller developer or probe end-user; and 2) describe actual probe performance -- used by probe vendors.

These scenarios are functional descriptions of what happens when a machine inspects a part using a probe. The technical scenarios, how the functions are accomplished using SIM APIs, are described in section 4, Conformance Tests. These descriptions reference Figure I-2 NGIS Machine Controller Model. The abbreviation "CE" will be used for "controller executive".

3.1 Point measurements.

Single points on a part are measured by moving a probe near the interesting area, then moving it to an *approach point* that is on a line normal to the surface of the part. The distance between the approach point and the part surface is the *probe approach distance*. The probe is then moved toward the part at the *probe approach rate* until the probe signals the motion system that it has sensed the part. The controller executive must capture machine axis and probe data, and halt probe motion before it collides destructively with the part.

3.1.1 Point measurement using switching probes

Switching probe behavior can be implemented in at least two ways:

- by a true touch-trigger probe with electrical contacts that open or close when the probe is deflected, or
- by a proportional probe that has been programmed to issue a binary signal when the part has been detected at some distance from the probe.

To completely specify performance of a switching probe for point measurements, the following parameters must be described:

Test parameters

- probe approach distance _____ mm .
- probe approach rate _____ mm/sec.

Probe characteristics

- probe "compliance", _____ mm (distance the probe tip can be moved toward the part, after the probe has issued its binary signal, without destructive collision with the part).
- time response of the probe, _____ micro-seconds (time to detect the part and the SIM to issue the binary signal).
- repeatability of probe triggering _____ micro-meters.

3.1.2 Point measurement using proportional probes

During the probe approach to the nominal point, the probe must generate new distance data and the controller executive must poll the data at a rate sufficient for probe motion to be stopped before destructive collision with the part. In robotics this is called a *guarded move.*

Scenario:

- CMM guides the probe toward the nominal point. Concurrently,
- Controller executive is accessing distance data from the SIM at a rate suited to probe approach rate and machine control dynamics.
- The part comes within the range of the probe. Then, the probe's distance value falls below a set threshold. Then the controller executive begins to halt motion.
- The controller executive accesses data from the SIM and the machine axis positions. At least two ways to capture probe and machine axis data can be used:
  - the motion controller adjusts the probe-to-workpiece distance to take advantage of highest probe accuracy. When probe motion has stopped, the CE accesses new probe distance data and captures machine axis position data, or

- the controller executive signals the SIM to capture its data before motion has stopped using a sync bus signal. The CE must simultaneously capture machine axis position data.

To completely specify performance of a proportional probe for point measurement, the following parameters must be described:

Test parameters

- probe approach distance _____ mm
- probe approach rate _____ mm/sec

Probe characteristics

- probe range _____ mm (distance that probe tip can be moved toward the part, after the probe produces its first useful distance data, without destructive collision with the part).
- time response of the probe, between a command to the SIM to generate new distance data, and availability of new data to the controller executive,

  _____ micro-seconds
- accuracy of probe _____ micro-meters

3.2 Scanning measurements

Scanning measurements of part features are made by gathering arrays of simultaneously generated data from a proportional probe and from the machine axis sensors. The machine controller must be able to discern from SIM data, if the probe is close enough to the part to produce useful distance data. For displacement-measuring probes, "close enough" means that the probe is touching the part. For proximity probes "close enough" means that the part lies within the sensing range of the probe and that the probe yields a desired accuracy within that range.

Note - The safe placement of a probe close enough to a part before the start of a scan can require the functionality of guarded move described above in 3.1.2 Point Measurements Using Proportional Probes.

3.2.1 Data synchronization issue

A machine controller that employs a SIM is a distributed system: i.e., parts of the controller and probe system are implemented on separate processor cards in the PC backplane. High speed communications of commands between the controller executive and the SIM is mandatory for ensuring accurate scanning measurements. The sync bus supplies the only means of simultaneously conveying commands from the control executive to distributed parts of the machine controller. If lower accuracy part measurements can be tolerated, then a scheme of commanding capture of new data using software interfaces controlled by a single CPU running the controller executive may be adequate. Here, the most important variables are speed of the CPU issuing software calls, speed of the communications interfaces used to convey the commands and data, and the physical probe speed in relation to the part.

3.2.2 Scanning with no machine servoing

This can only be accomplished when the geometry of the part is known well enough that a programmed probe path will not cause 1) the part going out of the probe's range, and 2) the probe to collide with the part.

Scenario:

- the probe is moved so that the part is in its measuring range.
- the controller executive begins sending simultaneous commands to the probe and machine axis sensor components to capture new data and to store it into FIFOs. The highest accuracy measurements will be produced by using the sync bus for issuing the commands.
- the controller executive executes a programmed path program while issuing the capture new data commands.
- at the end of the scan the controller executive stops issuing capture data commands, gathers the two FIFOs of data and stores them or does calculations to produce a single array of data points of probe center data or part surface data.

To completely specify performance of a proportional probe for scanning measurement, the following parameters must be described:

Test parameters

- probe scan rate _____ mm/sec.

Probe characteristics

- probe range _____ mm (distance that probe tip can be moved toward the part, after the probe produces its first useful distance data, without destructive collision with the part).
- time response of the probe, between a command to the SIM to generate new distance data, and availability of new data to the controller executive. _____ micro-seconds.
- maximum data capture frequency of the SIM and probe _____ Hz.

3.2.3 Scanning with machine servoing

In this case the controller executive must gather current probe data fast enough to control motion of the probe to keep the part within range. This could be needed when part geometry is not known beforehand, if part geometry differs from the expected geometry, or if the expected part location is not the same as the actual part location. During the scan two FIFOs of simultaneously captured data must be built, but probe distance data must also be conveyed to the controller executive in time for it to control probe motion.

To specify probe performance for this task the following parameters must be specified:

Test parameters

- probe scan rate _____ mm/sec.

Probe characteristics

- probe range _____ mm (distance that probe tip can be moved toward the part, after the probe produces its first useful distance data, without destructive collision with the part).
- time response of the probe, between a command to the SIM to generate new distance data, and availability of new data to the controller executive. _____ micro-seconds.
- maximum data capture frequency of the SIM and probe _____ Hz.

3.2.4 Free running rate

If the probe has an autonomous, free-running mode, where it generates capture-data commands internally, the fastest running rate must be specified by the probe vendor. With the SIM in this mode the controller executive 1) would not need to issue capture new data commands, and 2) reads SIM data which it assumes to be current. The controller executive must maintain the probe data FIFO. Note - this rate requires that the SIM do a new capture and conversion of probe data each time.

- Maximum free running data rate ______ Hz.

4. Conformance Tests

This section describes a technical scenario of SIM and probe testing, including initial integration of SIM hardware and software with a controller. This technical scenario describes how various measurement tasks are accomplished using API function calls. There are two types of testing:

- stand alone testing using a test configuration.
- operational testing using a machine controller.

There are ____ different tests. These test results, plus a SIM specification sheet should completely specify a probe's capabilities. Once a test is entered, each and every API call specified in the test must be executed. Only NGIS compliant function calls may be used.

4.1 Standalone testing using a test configuration

The configuration of a testbed for standalone SIM testing is shown in Figure A-1. The *test controller* module simulates a machine controller's Control Executive (CE). It also interacts with the person conducting tests through an *operator user interface* (UI). If the test controller is automated, it uses *test script files* that describe sequences of API function calls and parameters. The test controller may record results of tests, including function return values and data returned, in a *test result log*. The *sync bus hardware driver* plus *sync bus API* functions interface the test controller to the sync bus.

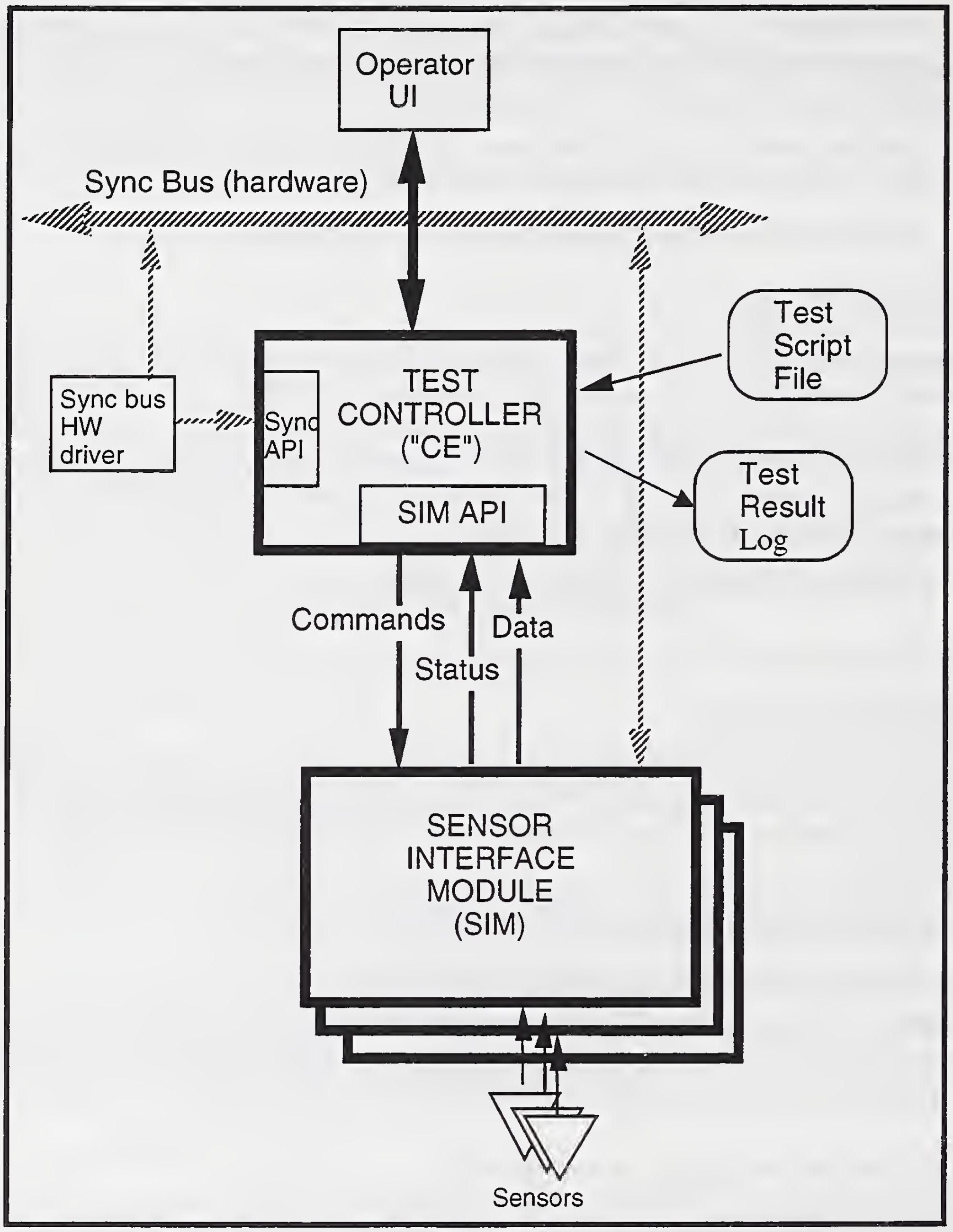


Figure A-1. Standalone Conformance Testing Configuration

4.1.1 SIM installation

- Description of computer platform (e.g., NT, ISA backplane, ...)
- Vendor Supplied Materials:
  - ♦ ____ SIM
  - ♦ ____ install disk that contains:

  - ____ NGIS compliant DLL, and .lib file (if appropriate)
  - ____ .ini file/s (optional)
  - ____ NT drivers (e.g. for dual port ram, serial port, ...)
  - ____ batch file for automatic installation
  - ____ SIM specification sheet (to specify bit masks for probe axes)

- Installation Procedure:
  - ♦ ____ Hardware install
  - ♦ ____ wiring to sync bus
  - ♦ ____ software install
  - ♦ ____ Linking of DLL
    - ____ static link (compile time link of .lib file)
    - ____ dynamic link (run time link of DLL)

4.1.2 Standalone testing of a SIM

Each test is a sequence of API function calls. The arguments used with the APIs and the return values and data must be recorded during a test.

A. BASIC SETUP AND CONFIGURATION OF SIM (mandatory)

This is a mandatory test for establishing communications with the SIM and for configuring it for operation. It must precede all other tests.

#	Return	Status	API func	arg1	arg2	arg3	arg4
1	________	________	SMRestart	COLD			
2	________	________	SMStrap	________			
3	________	________	SMLoad	"filename"			
4	________	________	SMConfig	"filename"			
5	________	________	SMCreateSimTask				
6	________	________	SMStartSimTask	simId			
7	________	________	SMAlive				
8	________	________	SMAlive				
9	________	________	SMName	______			

			String ret	"______"			
10	______	______	SMCreateSimTask	______			
11	______	______	SMStartSimTask	______ _			
12	______	______	SMErrorMsg	0			
			String ret	"______"			
13	______	______	SMErrorMsg	1			
			String ret	"______"			
14	______	______	DCSetup	______	______	______	
15	______	______	DCQueryFIFORecordLength				
16	______	______	DCgetMaxFifoSize	______			
17	______	______	DCallocateFifoSize	______			
18	______	______	MONgetData	______	______	______	______
19	______	______	MONgetStatus				
20	______	______	MONisExecuting				

B. SOFTWARE TRIGGERING OF DATA CAPTURE BY SIM

This test exercises software triggering of data capture, SIM building of it's FIFO, and accessing of the FIFO by the controller executive.

#	Return	Status	API function	arg1	arg2	arg3	arg4
1	______	______	DRgetNumRecords				
2	______	______	EVcommandedStore				
3	______	______	EVcommandedStore				
4	______	______	EVcommandedStore				
5	______	______	DRgetNumRecords				
6	______	______	DRretrieveRecords	______	______	______	
			Data	______	______	______	
				______	______	______	
				______	______	______	
7	______	______	DRgetNumRecords				

8	______	______	EVcommandedStore				
9	______	______	EVcommandedStore				
10	______	______	DRgetNumRecord				
11	______	______	DRretrieveRecords	______	______	______	
			Data	______	______	______	
				______	______	______	

Optional performance Test B1

This is a test to see how fast the probe can capture new data when commanded by software trigger. The controller executive calls EVcommandedStore as fast as possible. It waits for successful completion of each call.

- _____ number of calls
- _____ elapsed time (measured by CE)
- _____ data capture frequency of SIM (Hz)

C. SYNC BUS TRIGGERING OF DATA CAPTURE BY SIM

This test exercises SIM response to sync bus trigger events.

#	Return	Status	API Function	arg1	arg2	arg3	arg4
1	______	______	EVsetTriggerStore	______	______		
2	______	______	DRgetNumRecords				
3	The test controller causes a sync bus source to issue one or more sync pulses:						
	__________	sync bus freq (Hz)	_____	manual			
	__________	duration of pulses (sec)					
	__________	number of sync					

		pulses					
4	______	______	DRgetNumRecords				
5	______	______	DRretrieveRecords	______	______	______	
			Data	______	______	______	
				______	______	______	

D. SIM SYNC BUS OUTPUT - SWITCHING PROBE TRIGGERING

This test shows that the SIM can produce sync bus events when a switching probe detects the part.

#	Return	Status	API Function	arg1	arg2	arg3	arg4
1	Operator ensures that probe is not in prox. to part						
2	______	____	OUTsetOutput	______	______		
	_*Working Group Note - more needed here*_____	____					
	______	____					

E. SIM SYNC BUS OUTPUT – Proportional probe as switching probe.

This test shows that a SIM configured with a proportional probe can be programmed to emulate a switching probe's behavior described in test D.

#	Return	Status	API Function	arg1	arg2	arg3	arg4
1	Operator ensures that probe is not in prox to part						
2	______	______	EVsetThresholdStoreProcessed	______	______	______	______
3	______	______	OUTsetOutput	______	______		
4	Operator moves probe so that the probe senses prox or contact						
5	Test controller, by software or	______					

	oscilloscope, checks for issuance of sync trans.						

....

....

Z. ERROR MESSAGES

The SIM specification sheet must describe all possible error codes returned by API function calls and their meaning. It must also describe which error codes are supported by error message strings.

#	Return	Status	API Function	arg1	arg2	arg3	arg4
1	______	______	SMErrorMsg	_____	_____		
			String data returned:	" _____ "			
2	______	______	SMErrorMsg	_____	_____		
			String data returned:	" _____ "			
3	_____	_____	SMErrorMsg	_____	_____		
			String data returned:	" _____ "			
4	_____	_____	....				

4.2 Operational Testing using a machine controller

- Load DLL: run-time loading of dll (controller is in middle of an inspection task and a new probe is called for)
- Load DLL: static loading of dll (perhaps all DLLs are loaded/linked before inspection has begun).
- Non-operational tests (no machine motion, excite probe with artifacts or manually).
  - ♦ specify list of dll function calls and their order.
- Operational tests (doing inspection with motion control).

- touch-trigger probing
- scanning inspection
- sample part and operations.

5. SIM and Probe Performance Requirements

The data capture rate of individual SIM/Probes can be evaluated and specified using the following tests. The metrics are used by probe buyers to decide if the probe meets their measurement needs. Higher data rates do not necessarily indicate a higher quality probe - some measurement tasks may not require the higher data rates.

- digital to analog conversion time
- response to Sync bus

The Working Group is deciding if performance tests are within the scope of this specification.

6. Testing Glossary

probe[2]
"a device that establishes location of the movable components of a CMM relative to a measurement point. "

displacement-measuring probe [2]
"a probe that gives a signal proportional to a displacement of the probe from its free position."

proportional probe [2]
"a probe that gives a signal proportional to a distance between a reference point on the machine ram and the workpiece. Such probes may be displacement-measuring probes, proximity probes, or nulling probes"

proximity probe [2]
"a probe that gives a signal proportional to the distance from the probe tip to the workpiece."

probe approach distance [2]
"the distance of approach to the part at which the machine traverse speed is reduced to the probe approach rate for measurement"

probe approach rate [2]
"the nominal speed of approach of the probe toward the part during the acquisition of data (used primarily for switching probes)"

ram [2]
"the moving component of a machine that carries the probe"

switching probe [2]
"a probe that gives a binary signal as a result of making contact with, or being in proximity to, a workpiece. For this specification the only binary signal is transition of a sync bus level. "

validation [1]
"The process of evaluating a system or component during or at the end of the development

process to determine whether it satisfies specified requirements. "

verification [1]

"The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase."

verification and validation (V&V) [1]

"The process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements. "

Glossary References

[1] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610.12-1990.

[2] Methods for Performance Evaluation of Coordinate Measuring Machines, ASME B89.4.1-1997. Note - these definitions are quoted exactly from the standard. NGIS probes may not give signals that are mathematically exactly *proportional* to the distance to the workpiece - they may yield nonlinear signals that must be calibrated to give distance.

NGIS II SIM Specification

Appendix B. Discussion of SIM Communications Technologies

1. Equipment Requirements

The NGIS II SIM API is currently specified to operate on a PC computer, having an ISA bus, running Windows NT 4.0™. Other platforms may be supported in the future.

2. Candidate Software Technologies

The SIM Object Model was defined using the concepts of interfaces, inheritance, and data hiding. The first implementation of the API focuses on DLL technology, which does not have all of the constructs of object oriented modeling. The DLL technology is one implementation mechanism for realizing the functional interfaces defined by the Object Model. We anticipate that the functional interfaces to SIMs will be transitioned into more advanced technologies like OLE in the future.
Microsoft Communication Mechanisms

- DLL - dynamic link library
- DDE - dynamic data exchange
- OLE-Active X - Object linking and embedding

2.1 Dynamic Data Exchange (DDE)

DDE allows a set of predefined data to be available to other processes. The DDE architecture is a client/server architecture. A particular set of tag points is identified by the developer of the DDE Server. The DDE Client requests the token for particular names. The DDE Client then requests, on demand, the data with that token and receives the current value from the server.

2.2 Dynamic Link Libraries (DLL)

DLL are a set of code modules where each module contains a set of functions and/or data. DLLs contain a table of pointers to all functions and data contained in the DLL. Windows provides an application API for:

- Loading the DLL
- Obtaining the address for any function
- Obtaining the address for any piece of data
- Calling any function in the DLL
- Accessing any piece of data in the DLL
- Unloading the DLL

An application that requires a DLL will ask for it to be loaded. When a particular function or piece of data needs to be used, the application requests a function or data pointer for that item. The application can then call the function pointer or directly access that piece of data. When an application no longer needs a particular DLL, it can request that it be unloaded.

2.3 Object Linking and Embedding (OLE) and Active X

OLE and Active X represent a group of technologies that enable and facilitate component integration and component software. OLE offers extensible standards and mechanisms to enable software developers to package their functionality and content into reusable components. OLE is targeted at the integration of binary components independent of the language used to create those components.

The core of OLE technology was designed to address the shortcoming of previous technologies such as DLLs. This subset of the technology was originally called OLE Components and is now Active X Components. The shortcomings of the DLL that Active X addresses are as follows:

- No Enforced Type Checking
- Version Control is not supported
- Direct Access to Data is allowed

Once a DLL has been loaded, the programmer uses an assumed interface for the particular routine call that is desired and calls it. There are no checks to see if the parameters were the right size and type. With the OLE component technology, the parameters for every function are checked at the time the procedure is called.

Version numbers of a software technology are an attempt to represent the differences between functionality. OLE attempts to combine the features of a component interface into feature sets which it defines as interfaces. These feature sets can then be requested by a user of the component. New interfaces can be added without affecting the previous interfaces. A client must understand an interface and ask for it specifically before it can use that interface. You must request it to get it.

3. Scenario of SIM use.

A controller configured with three different SIMs and their DLL drivers is depicted in Figure B-1.

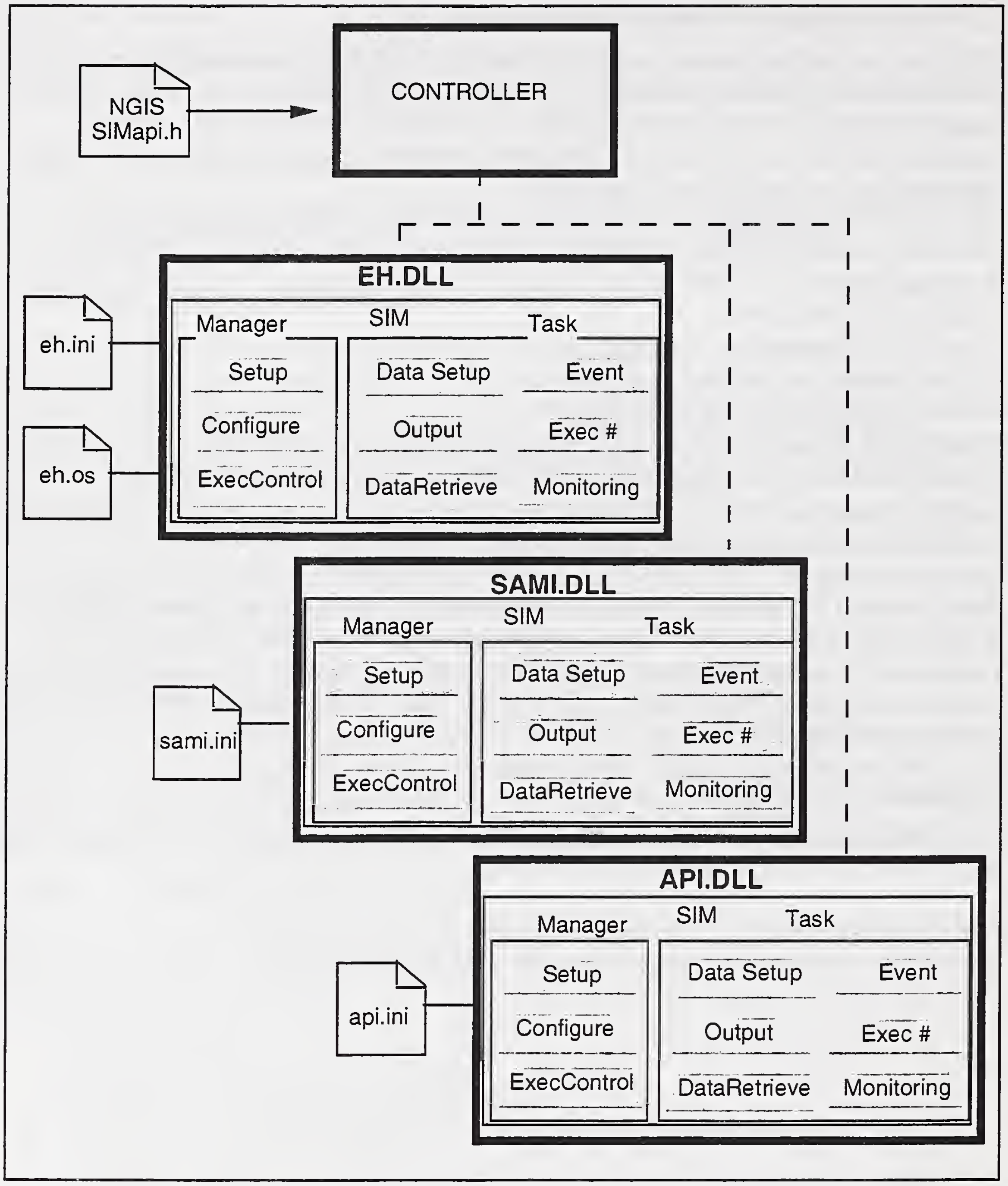
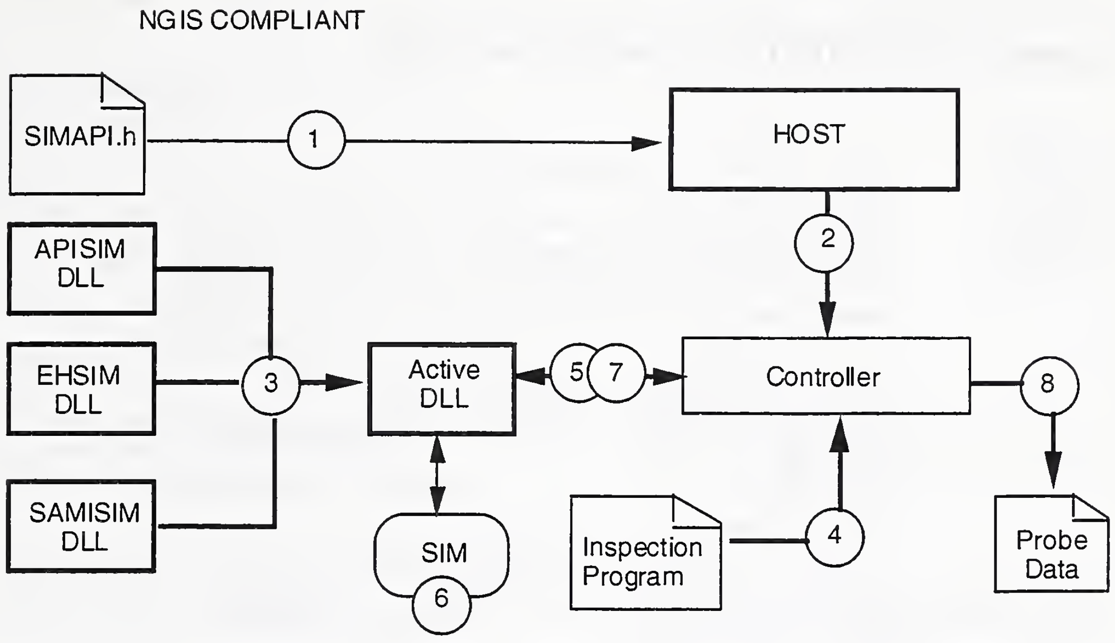


Figure B-1. Multiple DLLs in a controller

Figure B-2 shows the sequence of actions by the controller Host and Controller Executive to add SIM DLLs to the controller and to configure the SIM for inspection operation.



1. Controller vendor incorporates SIMAPI.h into product
2. Controller is compiled with hooks to SIM routines for dynamic loading
3. Controller allows for loading of the designated SIM.dll and loads it.
4. Inspection program statement calls for a probing operation.
5. Controller calls routines in the loaded dll to configure the SIM.
6. The SIM responds to sync pulses and stores the probe data in a FIFO.
7. Upon completion of the probing controller calls SIM routines to get the array of probe data.
8. Controller outputs data.

Figure B-2 . Actions by the controller Host and Controller Executive to add SIM DLLs to the controller and to configure the SIM for inspection operation.

Appendix C. SIM API DLL Specification (sim.h)

```
// "sim.h"
// Author:      NGIS Sensor API Working Group
// Date:        January 18, 1998
// Project:     NGIS II-SIM specification
// Notes:       The following file represents the DLL version of
//               the API
//              generated by the NGIS II SIM API Working Group.
//              The function calls are "flattened"-- they are not
//              object-oriented.
//              The SIM DLL macros allow for both
//              the SIM developer and user to compile against the
//              same file.
//              The developer adds the __SIM_DLL__ definition to
//              their code, e.g.,
//              at the top of the APIDLL.C file.
// Rev Date: Original by Nat Frampton, ATR Corp, 10/01/96.
// Rev Date: 11/6/96, comments by Eun Soo Lee of Automated
//              Precision, Inc.
// Rev Date: 5/30/97  ed. by Bill Rippey, result of working
//              group meeting of 5/28/97 at NIST.
// Rev Date: 21-Oct-97, cleanup with latest document edits,
//              Rippey.
// Rev Date: 18-Jan-98, ed. by Rippey, result of working group
//              meeting of 18-Jan-98 at Ford.


#ifndef __SIM_H__
#define __SIM_H__


 // Standard DLL Macro Definitions for Borland and Microsoft
#ifndef __BORLANDC__      // ie Microsoft Visual C++
  #if !defined  (__SIM_DLL__)
  #define    SIMAPI __declspec(dllimport)
  #else
  #define    SIMAPI __declspec(dllexport)
  #endif
#else                    // Borland C++ preprocessor definitions
  #if !defined (__SIM_DLL__)
  #define    SIMAPI
  #else
  #define    SIMAPI _export
  #endif
#endif

// Includes
//
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
```

```
// #Defines
//

#ifndef ENABLE
#define ENABLE  1
#define DISABLE 0
#endif
#define MAX_ERROR_MESSAGE_SIZE 256

// Standard SIMerror values.  The rest are vendor defined.
#define SIM_SUCCESS                             0L

// Communications codes, 100-199
// Errors for all communications media
#define SIM_COMM_ERROR                        100L  // Unspecified error.
#define SIM_COMM_ERROR_WRITE                  101L  // Known error on write.
#define SIM_COMM_ERROR_READ                   102L  // Known error on read.

// Errors for serial communications
#define SIM_COULD_NOT_INIT_COM_PORT           105L     //
#define SIM_RS232_FAULT                       106L    // Generic error
#define SIM_COMM_ERROR_BAUD_RATE              107L    // Incorrect baud rate


// Error for laser probes
#define SIM_LASER_OFF                         300L  // Laser was not on,
// reason unknown.
#define SIM_LASER_KEY_OFF                     301L  // Manual enable was not
// switched on.

// Errors of bject sensing
#define SIM_OBJECT_OUT_OF_RANGE               400L  // object out of sensing
                                                    // range.
#define SIM_OBJECT_OUT_OF_RANGE_NEAR          401L  // cannot sense object,
// SIM estimates near.
#define SIM_OBJECT_OUT_OF_RANGE_FAR           402L  // cannot sense object,
// SIM estimates far.
#define SIM_OBJECT_TOO_NEAR                   403L  // can sense object near,
// measurement not valid.
#define SIM_OBJECT_TOO_FAR                    404L  // can sense object far,
// measurement not valid.

// Errors of function parameters
#define SIM_INVALID_FUNCTION_PARAMETER         500L // A param. of the func
// call was invalid.


// General errors
#define SIM_FUNCTION_NOT_IMPLEMENTED  900L



#ifndef    ulong
#define    ulong    unsigned long
#endif
```

```
typedef ulong  SIMerror ;

// Enumerations
//
typedef enum
  {
  COLD         = 1,
  HOT          = 2
  }  RESTART_TYPE;

typedef enum
  {
  DUAL_PORT_RAM     = 1,
  IO_BUS            = 2,
  SERIAL_PORT       = 3,
  PARALLEL_PORT     = 4
  }  STRAP_TYPE;

// Sync bus  level and transition descriptions.
typedef enum
  {
  LEVEL_0        = 1,            // i.e. the level is 0.
  LEVEL_1        = 2,
  TRANS_RISE     = 3,            // i.e. the level changes from 0 to 1.
  TRANS_FALL     = 4
   }  TRANS_TYPE;

typedef enum
  {
  GREATER_THAN     = 1,
  LESS_THAN        = 2,
  GREATER_EQUAL    = 3,
  LESS_EQUAL       = 4,
  EQUAL            = 5
  }  COMPARE_TYPE;

typedef enum
  {
  DONE             = 1,
  EXEC             = 2,
  FAIL             = 3,
  PAUSE            = 4,
  ABORT            = 5,
  IDLE             = 6
  }  STATUS;


typedef enum
  {
   NO_ERROR         = 0,
   ESTOP            = 1,
   ABORTED          = 2,
   NOT_SUPPORTED    = 3
   }  ERROR_TYPE;


// Structures
```

```
//

// ----------------------------------------------
//
// SIM enumerated definitions for:
//
//        - transition types,
//        - compare operators,
//        - channels,
//        - manufacturer's channels (including sync bus).
//
// --------------------------------------------------------
// Structures
//
// Correspondence of channel numbers to sensor axes is defined
// by the manufacturer in the SIM data sheet.

 typedef struct CHANNELS            // This is bit field definition
{
  unsigned    Channel0   :1;        // LSB
  unsigned    Channel1   :1;
  unsigned    Channel2   :1;
  unsigned    Channel3   :1;
  unsigned    Channel4   :1;
  unsigned    Channel5   :1;
  unsigned    Channel6   :1;
  unsigned    Channel7   :1;
  unsigned    Channel8   :1;
  unsigned    Channel9   :1;
  unsigned    Channel10  :1;
  unsigned    Channel11  :1;
  unsigned    Channel12  :1;
  unsigned    Channel13  :1;
  unsigned    Channel14  :1;
  unsigned    Channel15  :1;
  unsigned    Channel16  :1;
  unsigned    Channel17  :1;
  unsigned    Channel18  :1;
  unsigned    Channel19  :1;
  unsigned    Channel20  :1;
  unsigned    Channel21  :1;
  unsigned    Channel22  :1;
  unsigned    Channel23  :1;
  unsigned    Channel24  :1;
  unsigned    Channel25  :1;
  unsigned    Channel26  :1;
  unsigned    Channel27  :1;
  unsigned    Channel28  :1;
  unsigned    Channel29  :1;
  unsigned    Channel30  :1;
  unsigned    Channel31  :1;            //

}  CHANNELS;                            // 32 Bits
```

```
 typedef struct BINARY_CHANNELS     // This is bit field definition
{
  unsigned    SyncF     :1;
  unsigned    Sync0     :1;
  unsigned    Sync1     :1;
  unsigned    Sync2     :1;
  unsigned    Sync3     :1;
  unsigned    Sync4     :1;
  unsigned    Sync5     :1;
  unsigned    Sync6     :1;
  unsigned    ManBit0   :1;
  unsigned    ManBit1   :1;
  unsigned    ManBit2   :1;
  unsigned    ManBit3   :1;
  unsigned    ManBit4   :1;
  unsigned    ManBit5   :1;
  unsigned    ManBit6   :1;
  unsigned    ManBit7   :1;
  unsigned    ManBit8   :1;
  unsigned    ManBit9   :1;
  unsigned    ManBit10  :1;
  unsigned    ManBit11  :1;
  unsigned    ManBit12  :1;
  unsigned    ManBit13  :1;
  unsigned    ManBit14  :1;
  unsigned    ManBit15  :1;
  unsigned    ManBit16  :1;
  unsigned    ManBit17  :1;
  unsigned    ManBit18  :1;
  unsigned    ManBit19  :1;
  unsigned    ManBit20  :1;
  unsigned    ManBit21  :1;
  unsigned    ManBit22  :1;
  unsigned    ManBit23  :1;

}  BINARY_CHANNELS;                         // 32 Bits



// The NGIS dll sensor API uses the C naming convention.
#ifdef  __cplusplus
extern "C" {
#endif

// SIMManager -- SIMManager -- SIMManager -- SIMManager --
//

SIMAPI      char * SMgetManufacturersId();
// String is the probe maker's name,  e.g., "Extrude Hone", "API".
SIMAPI      char * SMgetModelNumber();
// String is probe's model number,  e.g., "SAMI-MIDAS 123"
SIMAPI      double SMgetRevisionCode();
// Get a floating point number representing code version.
SIMAPI      char * SMgetSerialNumber();
// String is the serial number of the probe, e.g. "EAS 552885"
SIMAPI      char * SMgetDateCode();
// Returns release date of dll code - "mm/dd/yy-hr:min"
```

```
SIMAPI      char * SMgetProductDescription();
// Returns brief blurb to describe the probe.
SIMAPI      char * SMgetLicenseOwner();
// Returns name of liscense owner.

SIMAPI    SIMerror   SMStrap  ( STRAP_TYPE strap,
                                long       address );
    // Establish the physical communications interface to SIM.

SIMAPI    SIMerror   SMLoad    ( char * fullpathfilename );
    // Load & run the SIM executive.

SIMAPI    SIMerror   SMRestart  ( RESTART_TYPE  reset );
    // Restart SIM Manager -- return 0 for success.

SIMAPI    ulong      SMAlive    ( void );
   // Query SIM Manager to see if it is running - returns
   // heartbeat number.

SIMAPI    SIMerror   SMName     ( char * name );
   // Queries name of SIM manager, typically returns probe model.

SIMAPI    void       SMErrorMsg  ( SIMerror errnum,
                                   char* pErrorMsg );
   // Writes ASCII text of error message to pErrorMsg. If no
   // message, writes null string.

//
// CONFIGURATION

SIMAPI    SIMerror    SMConfig ( char * inputFilename);
   // Configure the SIM using parameters in named file.
//
// SIM TASK MANAGEMENT

typedef long  simId ;

SIMAPI    simId     SMCreateSimTask     ( void );
   // Return 0 if could not create.
SIMAPI    SIMerror  SMStartSimTask      ( simId id);
SIMAPI    SIMerror  SMStopSimTask       ( simId id);
SIMAPI    SIMerror  SMRestartSimTask    ( simId id);
SIMAPI    SIMerror  SMDeleteSimTask     ( simId id);


// SIMTask -- SIMTask -- SIMTask -- SIMTask -- SIMTask
//
// SimTask Method Prefixes:
//  Config :    CFG. None in level 1, configuring done globally
//                in manager
//  DataSetup : DC
//  Event :     EV
//  Exec :      EX  - single versus blocks of data storage
//  Output (to hardware) : OUT
//  Monitor :  MON
//  Data Retrieval : DR
// DATA SETUP
// ----------------------------------------------------

SIMAPI  SIMerror  DCsetup(CHANNELS         raw,
                          CHANNELS         processed,
                          BINARY_CHANNELS  manufacturer_data );
   // Specify the data channels to convert.

SIMAPI   int       DCqueryFIFORecordLength( void );
   // Returns number of bytes in one FIFO record (one entry).
   // e.g. if 3 data channels are specified, the FIFO record
   // length is 12 bytes.

SIMAPI   SIMerror  DCallocateFIFOSize( int num_bytes );
   // Command the SIM manager to allocate "num_bytes" bytes
   // for FIFO data storage.

SIMAPI   int       DCgetMaxFIFOSize( void );
   // Query the max size allocated for the FIFO in bytes.

//
// EVENT TRIGGER PROGRAMMING

SIMAPI  SIMerror  EVsetTriggerStore(
                                BINARY_CHANNELS         SyncEvents,
                                TRANS_TYPE              type ) ;
   // Define a sync bus event as a trigger.

SIMAPI   SIMerror  EVsetCounterStore(
                                ulong timer_num,
                                ulong count_down_ms );
   // Program a periodic elapsed time trigger -- units are
   // milliseconds.

SIMAPI   SIMerror  EvsetThresholdStoreRaw(
                                       CHANNELS        channel,
                                       COMPARE_TYPE  comparison,
                                       ulong           value,
                                       ulong           reset);
   // Define a trigger: when a processed data channel crosses a
   // threshold.

SIMAPI   SIMerror  EVsetThresholdStoreProcessed(
                                       CHANNELS        channel,
                                       COMPARE_TYPE    comparison,
                                       float           value,
                                       float           reset) ;
   // Define a trigger: when a processed data channel crosses a
   // threshold.

SIMAPI   SIMerror  EVcommandedStore( void );
```

```
// OUTPUT
// -----------------------------------------------------


SIMAPI    SIMerror  OUTsetOutput(
                               BINARY_CHANNELS zero_channels,
                               BINARY_CHANNELS one_channels,
                               BINARY_CHANNELS toggle_channels);
   // Program outputs as responses to triggers.


// EXEC
// ---------------------------------------------------------

SIMAPI    SIMerror  EXsetNumberExec(int count);
   // Program SIM to respond to a maximum of count triggers.

SIMAPI    SIMerror  EXContinuous( void );
   // Program SIM to respond to triggers until the FIFO is full.

// MONITOR

SIMAPI  SIMerror MONgetData(
                        CHANNELS        raw_channels,
                        CHANNELS        processed_channels,
                        BINARY_CHANNELS manufacturers_channels,
                        void            * loc,
                        int             record_size_match_check);

// Convert the specified data channels and write
// "record_size_match_check" bytes to
// the user's address "loc".
// NOTE -- MONgetData does not produce a FIFO entry or change
// any entries in the FIFO.

SIMAPI  STATUS     MONgetStatus(void);
   // A general purpose return of status.

SIMAPI  ERROR_TYPE MONgetError( void );
   // If SIM Task aborted, returns error number.


// DATA RETRIEVAL
// -------------------------------------------------------

SIMAPI  int  DRgetNumRecords( void );
   // Returns the number of data entries in the FIFO.

SIMAPI   SIMerror  DRretrieveRecords(
                               int    num,
                               void * loc,
                               int    record_size_match_check);

   // Retrieves num entries from the FIFO  into loc.
   // Number of stored entries in FIFO is reduced to reflect this
   // retrieval.  e.g., if the number of FIFO records
   // is 5, and 3 are retrieved, the number of records should now
   // equal 2.
   // record_size_match_check is the size of one FIFO entry, in
   // bytes.

#ifdef __cplusplus
}
#endif

#endif       //  __SIM_H__
```

NGIS II SIM Specification

Appendix D. SIM State Diagram

The concepts for this section were discussed at the 18-Jan-98 working group meeting. It has not been reviewed in detail by the group, but is included here to reflect the working group's latest efforts.

This section describes the state transitions of an NGIS SIM. The transitions are caused by external actions and internal events. External events are application program invocation of API methods and sync bus events. Internal events are periodic SIM timers, input channel thresholds, and internal SIM actions, e.g. resulting in state transition from EXECUTING to DONE or from EXECUTING to ERROR.

Figure D-1 illustrates the sequence of states and state transitions that a typical NGIS controller would encounter. Upon startup the controller is in the POWERUP state. This state assumed that the SIM hardware also has power.

The *SMLoad* method causes a state transition from the POWERUP state to the LOADED state. The action associated with this state transition would is to load the SIM executive from disk into the SIM RAM. This state transition may be optional if the SIM supports a ROM kernel, however, all SIMs must support this state transition even if no kernel loading from disk is necessary.

In the LOADED state the Controller Executive can invoke the *SMStrap* method to establish hardware communications with the SIM. The method *SMRestart* causes the state transition from the LOADED state to the READY state. The action associated with this state transition is for the SIM Manager to reset all the internal variables and clear the SIM Task. The SIM Manager can be commanded to import operating parameters from a disk file using the *SMConfig* method.

SMCreateTask causes the state transition from the LOADED to the CONFIGURING state. The action associated with this state transition is to create a new SIM Task that the controller can then configure. Four classes of methods can configure the SIM Task.

The *SMStartSimTask* method causes the transition to the RUNNING state, in which the SIM is actively monitoring to respond to external triggers, and to generate sync bus and output events. Monitor methods can be invoked while the SIM is RUNNING. The FIFO is accessed first by using *SMStopSIMTask*, and then *DRRetrieveData.* SMStartSimTask clears the FIFO and puts the SIM back into RUNNING. SMRestart causes the same state transition but preserves the FIFO.

All SIM parameters and configuring information persists until the SIM Task is deleted, using SMDeleteTask.

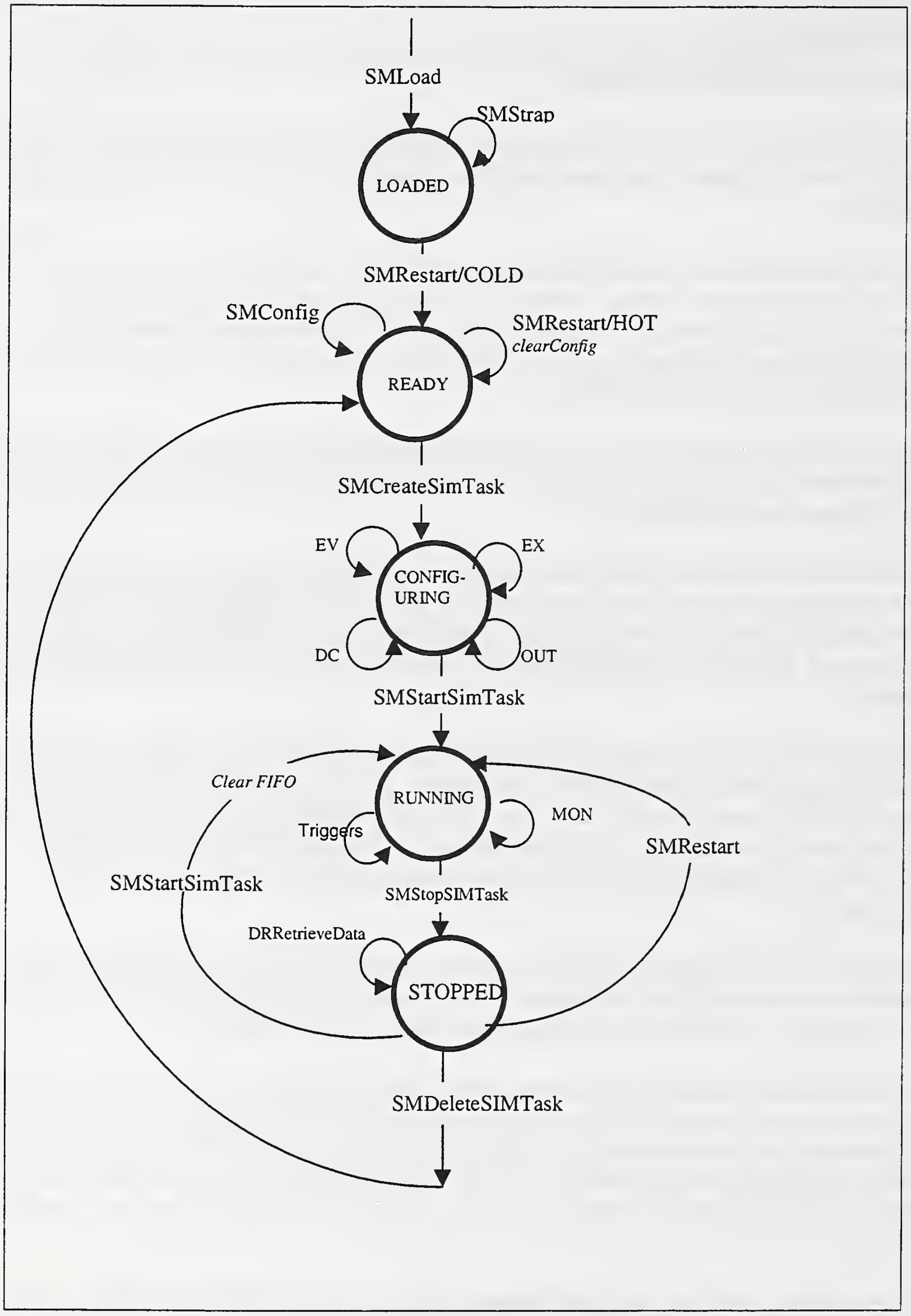


Figure D-1. SIM State Diagram