# A MODULAR SIGNAL PROCESSING SOFTWARE DEVELOPMENT SYSTEM

by

Peter A. Rigsbee
Naval Research Laboratory
Washington, D.C., U.S.A.

## ABSTRACT

A software development system is being built by the U.S. Naval Research Laboratory (NRL) to aid production of modular signal processing software for U.S. Navy airborne ASW platforms using the Navy's Advanced Signal Processor (ASP). This system, part of the ASP Common Operational Software (ACOS) project, is intended to reduce the costs of programming, maintaining, and modifying ASP software. The development system includes the SPL/I support software system, the ACOS Program Generator, and the ACOS Shell. SPL/I is a high-level language designed for real-time signal processing applications. CROS, the standard SPL/I operating system, supports the multiprogramming and data management features of the SPL/I language. SPL/I and CROS are being used by the Navy in signal processing applications. The ACOS Program Generator (APG) is the key component of the development system. It translates software written in a high-level, ASP-independent block language into SPL/I code to be processed by the SPL/I Compiler. This language allows the programmer to speak in terms of bulk storage variables, queues of data buffers, and parameterized signal processing primitives such as "FFT" or "AGC". The APG then takes the responsibility for generating or executing, as appropriate, code to manage storage and to initiate data transfers and complex functions; effectively hiding the underlying hardware architecture from the programmer. The ACOS Shell, which serves as an extension to CROS for the ASP, is the runtime key to the system. The Shell controls and manages access to ASP resources and is the runtime signal processing controller. The Shell is accessed only by software generated by the APG, and the programmer is never aware of the distribution of functions among CROS, the Shell, and the hardware.

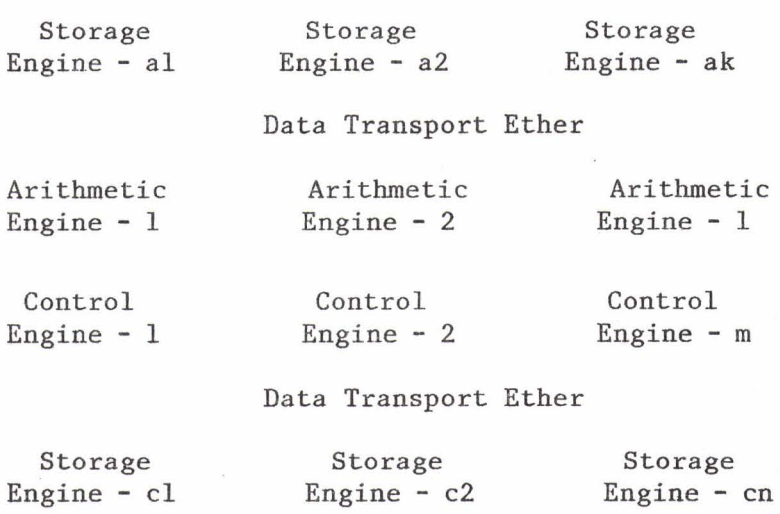
## ACOS PROGRAM GENERATOR FACILITIES

The ACOS Program Generator (APG) is a programming tool that allows the implementation of signal processing software in a target machine independent manner. The APG converts its input into SPL/I code that accesses CROS and the ACOS SHELL (see below) for services. The APG will be responsible for:

- Allocation of storage
- Transfer of data between storage areas
- Creation of control information for the SHELL
- Scheduling signal processing and data transfer operations

PROGRAM GENERATOR USER ARCHITECTURE

The ACOS user has the following view of a generic signal processor (GSP) architecture for which he is to write programs:




There is some number of independent Arithmetic Engines, each of which is used to perform signal processing operations. There is some number of Storage Engines which can be accessed by Arithmetic Engines via a Data Transport Ether. These Storage Engines are used to buffer incoming data, partially processed data, and outgoing data as needed. Arithmetic Engines cannot access other Arithmetic Engines. Any Arithmetic Engine-accessible GSP storage not in Storage Engines is considered to be local to an Arithmetic Engine. This local storage (if any) associated with an Arithmetic Engine is allocated by the ACOS system (APG and/or SHELL).

There is some number of independent Control Engines, each of which is used to perform control operations. There is some number of Storage Engines which can be accessed by Control Engines via a Data Transport Ether. Storage Engines are used to buffer incoming data, partially processed data, and outgoing data as needed by Control Engines to determine control actions. Control Engines effect control through a Control Ether (not shown in the diagram) that connects to all GSP engines. Control Engines cannot access other Control Engines except through the Control Ether. Any Control Engine-accessible GSP storage not in Storage Engines is considered to be local to a Control Engine. The local storage (if any) associated with a Control Engine is allocated by the ACOS system (APG and/or SHELL).

An ACOS user first writes a number of ENGINE PROGRAMs. ENGINE PROGRAMs are the basic units of software that are executed by the GSP. They specify control processing or signal processing to be performed by the GSP to meet operational requirements. The user specifies how ENGINE PROGRAMs are related to one another by defining a set of GRAPHs. These GRAPHs specify

the topological connections among ENGINE PROGRAMs (ENGINE PROGRAMs can be viewed as the nodes of GRAPHs) by identifying queues to be used to exchange data between pairs of ENGINE PROGRAM executions or between an ENGINE PROGRAM execution and the outside world. This user software, comprised of ENGINE PROGRAMs and GRAPHs, is processed by the APG and the SPL/I Compiler and loaded into the target hardware.

At runtime the GSP executes ENGINE PROGRAMs under the supervision of the SHELL. Each instantiation of an ENGINE PROGRAM is executed as an independent task by exactly one GSP engine. A GSP engine executes at most one task at a time. When a GSP engine completes execution of a task, the task and its associated local storage is destroyed. Two engines can concurrently execute distinct tasks that are instantiations of the same ENGINE PROGRAM. Tasks are created by the SHELL in response to the presence of data in the queues specified in GRAPHs. An instance of a GRAPH (called a GRAPH task-set) is created at runtime by STARTing the GRAPH. STARTing the GRAPH causes the SHELL to begin monitoring the GRAPH-specified queues. Upon discovering the presence of sufficient data in some of the queues, the SHELL will create tasks to execute GRAPH-specified ENGINE PROGRAMs.

Tasks are differentiated between control tasks and arithmetic tasks. Control tasks are executed by Control Engines and arithmetic tasks are executed by Arithmetic Engines. Only control tasks can start and stop GRAPHs. Consequently, ENGINE PROGRAMs are called control ENGINE PROGRAMs for arithmetic ENGINE PROGRAMs depending upon the type of GSP engine on which they are to execute. Similarly, GRAPHs are either control GRAPHs or arithmetic GRAPHs. Initially at runtime a user-specified control GRAPH is STARTed by the ACOS system. This initiation causes the SHELL to create a control task-set. This task-set may then proceed to start other control GRAPHs or arithmetic GRAPHs defined by the ACOS user.

The numbers of Arithmetic Engines, Control Engines, and Storage Engines may differ among specific implementations of the GSP. The Storage Engines that a particular Arithmetic Engine or Control Engine can access may also differ among specific implementations of the GSP. The engines are not necessarily homogeneous. Associated with each GSP engine is a set of capabilities that the engine can support. For example, an Arithmetic Engine may be capable of executing only certain arithmetic primitives. A Storage Engine may be capable of storing only variables of particular SPL/I modes. The particular capabilities associated with an engine depend upon the specific implementation of the GSP.

## PROGRAM GENERATOR USER FACILITIES

The definitions of the APG facilities are given below.

### Engine Identifiers:

Arithmetic Engine, Control Engine, and Storage Engine identifiers are of two types: engine designators and engine class designators. An

engine designator references a particular Arithmetic Engine, Control Engine, or a Storage Engine. An engine class designator references a specific set of Arithmetic Engines, Control Engines, or Storage Engines. Engine identifiers are used by the APG and the SHELL to assign execution and storage resources. In the event an engine class designator is used, then the ACOS system will select one engine from the specified set.

Storage Template Declarations:

FIELD

Defines size of field used in one or more block definitions. Size defined in bits.

BLOCK

Defines data template. FIELD and BLOCK pertain only to the logical layout of data. Entities declared using FIELD and BLOCK cannot be accessed using ordinary SPL/I features; thus SPL/I packing is neither assumed nor enforced. BLOCK templates are used to define the basic data entities that comprise instances of QUEUEs and of BUFFERs.

QUEUE STRUCTURE

Defines queue data template. QUEUE STRUCTURE names are used to define QUEUE variables and formal inputs and formal outputs of ENGINE PROGRAMs and GRAPHs. QUEUE STRUCTUREs represent FIFO queues of data items defined using a common BLOCK template.

Variable Declarations:

BUFFER

Defines the names for BUFFER variables. BUFFER variables exist only in Arithmetic Engine and Control Engine local storage; BUFFER variables are used for data exchange among PRIMITIVEs. A BUFFER variable has two components: control information and a data element (an instance of an underlying BLOCK template). Storage for control information may be allocated (when an ENGINE PROGRAM task is executed) and freed (when the ENGINE PROGRAM task completes) as a result of the BUFFER declaration. Data element storage is allocated and freed by execution of BUFFER operations listed below.

Operations on BUFFER variables:

CREATE – create storage for the data element of the specified BUFFER variable

REPLACE – write a PRIMITIVE output into the data element of the specified (and previously CREATED or INSERTed) BUFFER variable

INSERT – create storage for the data element of the specified BUFFER variable and write a PRIMITIVE output into it

REMOVE – read the contents of the data element of the specified BUFFER variable as a PRIMITIVE input and destroy the data element storage upon completion of the read

COPY – read the contents of the data element of the specified BUFFER variable as a PRIMITIVE input

DESTROY – destroy previously CREATED or INSERTed data element storage for the specified BUFFER variable

QUEUE

Defines the names for QUEUE variables. QUEUE variables can exist only in Storage Engines. QUEUE variables are used for data exchange among ENGINE PROGRAM tasks. A QUEUE variable has two components: control information and queue elements (instances of the underlying BLOCK template). Storage for control information is allocated (when a GRAPH is STARTed) as a result of the QUEUE declaration. Queue elements are allocated and freed by execution of QUEUE operations listed below.

Operations on QUEUE variables:

ENQUEUE – add specified number of queue elements to the tail of the specified QUEUE variable

DEQUEUE – remove and read specified number of queue elements that are at the head of the specified QUEUE variable

READQUEUE – read specified number of queue elements that are at the head of the specified QUEUE variable

Topological Declarations:

PRIMITIVE

Defines a low level primitive available to an ENGINE PROGRAM. A PRIMITIVE may consist of any number and combination of executions of lower level operations on an Arithmetic Engine or Control Engine. PRIMITIVE declarations are always provided a priori to the APG user.

ENGINE PROGRAM

Defines a sequence of SPL+I code and PRIMITIVE invocations that perform a particular processing algorithm. An instantiation of an ENGINE PROGRAM is called a (control or arithmetic) task and executes on a specified Control Engine or Arithmetic Engine. Any number of instantiations of an ENGINE PROGRAM can exist concurrently. QUEUE

variables are the inputs and outputs of tasks. A task is scheduled by the SHELL when the number of queue elements on each of its input QUEUE variables meets or exceeds the threshold specified for that QUEUE variable.

NODE

Associates actual QUEUE variables with formal QUEUE parameter names for a specified ENGINE PROGRAM.

GRAPH

References a set of ENGINE PROGRAMs and declares the QUEUE variables that serve as the inputs and outputs of ENGINE PROGRAM tasks. A GRAPH is a static description of how instantiations of ENGINE PROGRAMs are to exchange data. STARTing a GRAPH creates an instance of the GRAPH (called a GRAPH task-set) and causes the creation of control storage for the QUEUE variables declared within the GRAPH declaration and for the QUEUE variables declared within the ENGINE PROGRAMs referenced within the GRAPH declaration. There is no executable SPL/I code within a GRAPH; it is simply a collection of storage template, QUEUE, and NODE declarations. A GRAPH must reference only arithmetic ENGINE PROGRAMs or only control ENGINE PROGRAMs.

Primitive Statement:

INVOKE

When executed, specifies that execution of a PRIMITIVE should be initiated. The INVOKE includes specification of all the input and output variables required for initiation of a general purpose PRIMITIVE to meet a particular requirement. The QUEUE and BUFFER operations described above are used to access QUEUE and BUFFER variables. The Control Engine or Arithmetic Engine specified in the ENGINE PROGRAM declaration containing the INVOKE will execute the INVOKEd PRIMITIVE.

Control Statements:

START

When executed, specifies that an instance of a GRAPH (called a task-set) is eligible for scheduling. An already STARTed GRAPH task-set cannot be reSTARTed without an intervening STOP of the GRAPH task-set.

STOP

When executed, specifes that a GRAPH task-set is no longer eligible for scheduling. In the event a control task-set is STOPped, then all GRAPH task-sets STARTed by that control task-set are also STOPped.

Execution:

Runtime entities:

An instantiation of an ENGINE PROGRAM is called a (control or arithmetic) task and executes on a specified Control Engine or Arithmetic Engine. QUEUE variables are the inputs and outputs of tasks.

Task Queue Discipline:

A task is scheduled by the SHELL when the number of queue elements on each input QUEUE variable meets or exceeds the threshold specified for that QUEUE variable in the ENGINE PROGRAM's formal parameter list and in the referencing NODE declaration. By its completion a task must have DEQUEUEd exactly the number of queue elements specified for each input QUEUE variable as the consume amount and the task must have ENQUEUEd exactly the number of queue elements specified for each output QUEUE variable as the produce amount.

Computational Determinancy:

Any number of instantiations of an ENGINE PROGRAM can exist concurrently. In the event that an ENGINE PROGRAM's engine identifier is a class designator, then the SHELL preserves the queue data ordering implicit in each GRAPH for concurrent instantiations of the same task-set NODE. (It is sufficient to prohibit multiple concurrent tasks executing the same task-set NODE. It may be useful, however, to allow multiple concurrent tasks executing the same NODE in a task-set. If they do, then "the NODE" must be run as a pipe.)

## ACOS SHELL FACILITIES

The ACOS SHELL is the runtime support software of the ACOS system. Development of the SHELL facilities must, of necessity, lag behind that of the APG facilities. The following is a brief description of the main features of the ACOS SHELL.

### Monitors

The SHELL consists of a set of runtime engine monitors. Each engine (control, arithmetic, or storage) is represented by a single monitor. Monitors consist of SPL/I processes and procedures that have three functions:

- task scheduling: the initiation of the execution of an ACOS task.
- meassage transmittal: the format and transmittal of a message to some other monitor.
- message receipt: the translation and processing of a message sent by some other monitor.

## Lists

Monitors manage two logical lists of resources. One list is that of GRAPHs that can execute on the engine. This list contains information about all the tasks that comprise the GRAPH. The other list is that of QUEUEs stored in the engine. Not all monitors necessarily manage both list types, nor do the number of elements in the lists necessarily bear any relationship between monitors.

## Messages

Monitors communicate through the use of messages. The following is a tentative list of SHELL messages.

Message Processing Complete – sent after processing of some other message; specifies that processing has completed and that another message may be sent to this monitor.

Start Graph – specifies that a particular graph may be scheduled. May cause "Connect Queue" messages to be issued.

Stop Graph – specifies that a particular graph may no longer be scheduled, may cause "Disconnect Queue" messages to be issued.

Connect Queue – associates actual queues with a node, graph, or system feature as a sink or as a source.

Disconnect Queue – disassociates actual queues from a particular sink or source.

Add Data – transfers data to a queue. May cause "Data Ready" message to be issued to the queue sink.

Read Data – asks for data to be sent from a queue.

Data Transmit – sends data from a queue in response to a "Read Data" message.

Data Ready – indicates that a queue has sufficient data for task initiation.

## Error Processing

Monitors must monitor and process runtime error conditions. The set of error conditions and the processing mechanism is to be determined.

SUMMARY

The APG and ACOS SHELL are intended to raise the level of the programming signal processing software beyond that of standard high level languages. This will both reduce programming cost and, because of the improved understandibility of the resultant operational software, reduce life-cycle maintenance costs. Although designed for the ACOS project, it is of sufficient generality for use in other signal processing application. In addition, because architecture details are buried in the PRIMITIVE definitions and the SHELL implementation, the resulting operational software is highly machine transportable.

DISCUSSION

J.E. Vernaglia Will SPL/I programming still be necessary under the ACOS system?

P.A. Rigsbee Yes, SPL/I code is integrated with APG facilities in the ACOS system. Some functions will need none of the ACOS features and, therefore, will be coded purely in SPL/I.

R. Seynaeve What was the overall cost of SPL/I development?

P.A. Rigsbee Two and half million dollars and five years.