

GEDAE Technical Document #2

GEDAE Application Structure

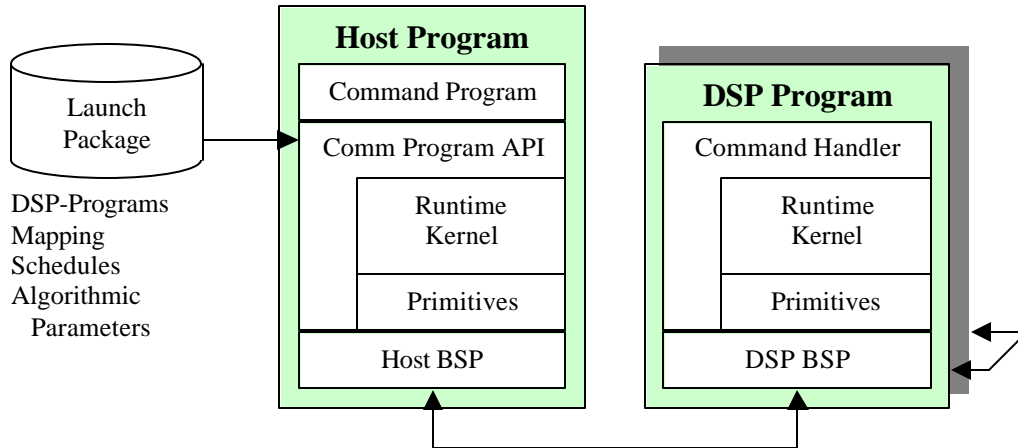
August 9, 2000
By Kerry Barnes

Building Applications with GEDAE

GEDAE provides a portable, scalable, and efficient framework for developing, executing, controlling and observing standalone application on embedded processors. This document describes how this is achieved. It begins by introducing the parts of a GEDAE standalone application. It then describes how the GEDAE development environment is used to produce an application specification, translate that specification into a standalone application implementation, and then test and observe the standalone application. The standalone application is examined in detail beginning with the parts of the application that the user specifies. It continues by describing the command program API, which allows developers to create portable command programs that can control every aspect of the application. The runtime kernel, which provides the distributed data flow control, is then described. Both host to DSP and DSP to DSP communication are described with emphasis on the efficiency of the flexibility and efficiency of the communication. Finally the Board Support package which provides the low-level framework on which a GEDAE application is built is described.

The GEDAE Application

A standalone GEDAE application, pictured below, begins execution as follows. The Host Program reads a description of the algorithm's implementation (launch package). It starts up the DSP programs and initializes them. The application then runs distributed on the host and the DSPs. While the host program has complete high level control of the application, the data flow control of the application is efficiently distributed across the Runtime Kernels. The different parts of the application are defined below:



GEDAE Application

Launch Package – The output of the GEDAE Development Environment that contains everything needed to run a standalone GEDAE application.

Command program – the top-level program that has control over all aspects of running the application. This can optionally be generated by the developer or by GEDAE.

Command Program API – the application program interface that provides the Command Program with control over all aspects of an application.

Command Handler – receives and handles commands sent to it by the command program to implement the various command program functions.

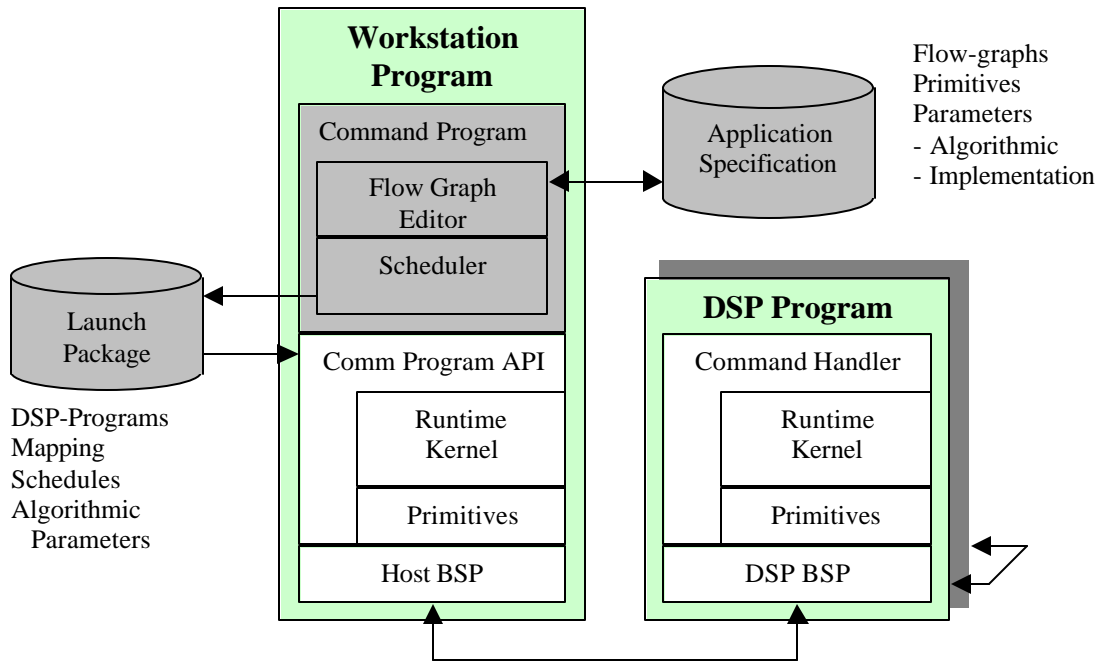
Primitives – the leaf function boxes that implement the basic algorithms in a data flow graph. Primitives are executed by the Runtime Kernel.

Runtime Kernel – the distributed schedule engine that runs the schedule of primitive executions needed to implement the algorithm captured by the development environment.

Board Support Package – The GEDAE Board Support Package provides a common interface to the hardware resources of each specific BSP.

Developing GEDAE Applications

The GEDAE Development environment can be considered a special case of a GEDAE application program that captures an application specification and translates this into a standalone GEDAE application. The software structure that executes the application is the same for the development environment and the standalone application so the standalone application will have the same performance as observed during development.



GEDAE Development Environment

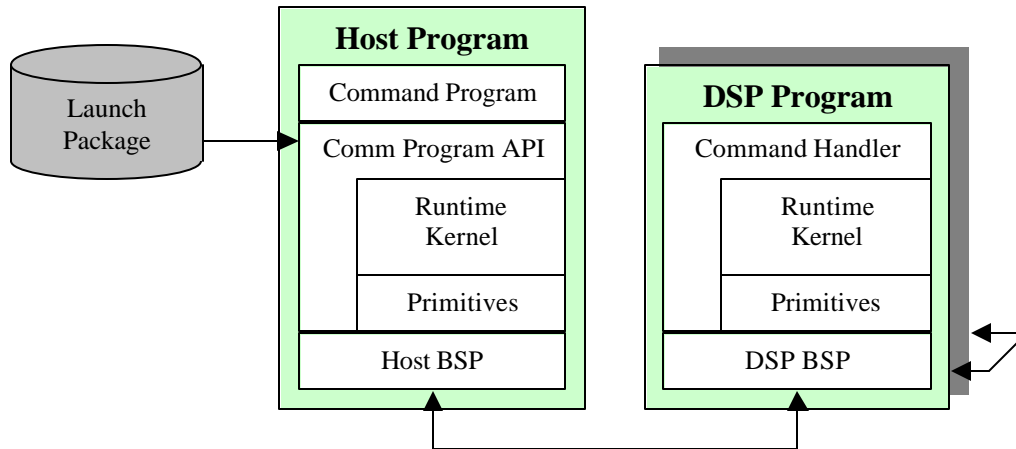
Two of the most important parts of the Development Environment are the Flow Graph Editor and the Scheduler.

The Flow Graph Editor provides the interface from which a user can capture the application specification. It lets users develop and test graphs using primitives from the extensible primitive library and optimize the graph's performance to run on multiple processors. It lets users monitor graph performance using the GEDAE Trace Table.

The Scheduler is the part of the GEDAE development environment that is responsible for translating the algorithm specification into a launch-package. The launch package created consists of DSP executables with primitive boxes that use optimized vector libraries on the target DSP. It also contains the schedule information required to execute the algorithm described by the GEDAE data flow-graph.

The Launch Package – The Application Implementation

The launch package is the output of the GEDAE development environment that provides the standalone application implementation. It is a self contained directory that can be archived and delivered and can be run on any platform that has the same hardware configuration as the system for which the launch package was developed.



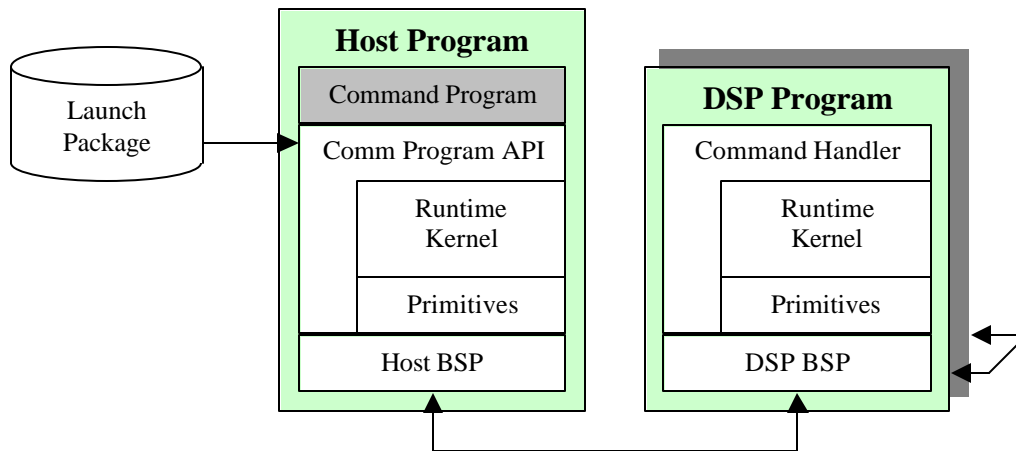
The Command Program provides high level control.

A launch package contains the following files

- exec-host – This is a simple automatically generated command program that can read the launch package and start it running. The exec-host also contains all of the primitive function boxes needed to implement the portions of the application mapped to the host and all of the host control boxes from the original graph. GEDAE provides a rich capability for embedding control in the flow-graph. For applications where all the control is part of the flow-graph the exec-host is all the command program that is needed.
- embedded executables – These executables implement the DSP program and contain the minimal set of capabilities needed to implement the application.
- schedule description files – This file describes the preplanned memory layout of the schedules on each processors in a compact format that can be easily downloaded to the processor.
- launch file - Contains the mapping of executables to processors.
- init.c – A file containing a code generated version of the parameter equations for this graph. The code generation allows derived parameters to be calculated efficiently.
- bindings – A description of all the parameters in the graph including derived parameters. Includes parameter names, type and dependencies.
- default – The default values for all settable graph parameters. It contains the parameter settings as of launch package creation but is easily modified by the user.
- synonyms – A file of synonyms for parameters and data streams that promotes portability of command programs.

Controlling the Application – The Command Program

The command program is the part of a GEDAE application that provides the high level control over an application. It is NOT responsible for scheduling the dataflow algorithms on each of the embedded processors. This is the responsibility of the Runtime Kernel which is distributed on many of the different processors.



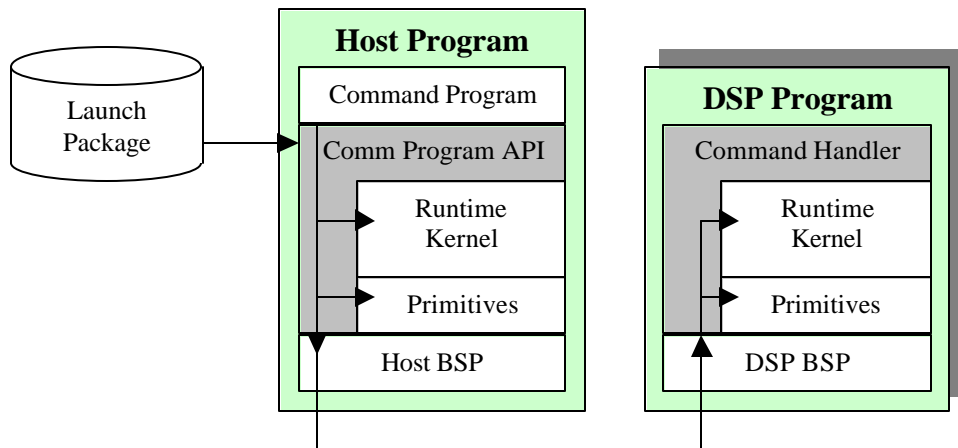
The Command Program provides high level control.

There are three basic types of command programs that the user will encounter:

- GEDAE Development Environment – When developing applications a user is actually running a command program that creates and dynamically links in its own launch package. Thus the user will achieve the same performance and behavior at development time as at runtime.
- Launch package automatically generated exec-host – When creating a launch package an exec-host routine is included that will run the launch package much as when the user clicks the “Run” button from the development environment. GEDAE allows sophisticated mode control and GUIs to be built directly into a graph so the exec-host is often the only command program needed. The exec-host also has command line arguments that allow a user to bind files to application data ports and to set trace levels on the command program execution.
- User generated command program – When the user wants control over a launch package that was not already built into it or wants to load and control multiple launch packages the user can generate a command program that does this. A customized command program can read multiple launch packages, re-map the launch packages at runtime and establish connections between running applications. It can also set parameter values and read and write data to the graph. In so doing it can provide an interface between resources that only the host has access to and the running application. Such access to resources can also be encoded in primitive function boxes. It's your choice to put the control in the command program or the graph.

Controlling an Application – The Command Program API

A user writes a command program using the Command Program API or COMMAPI. The 100+ functions of the COMMAPI provide the user command program with portable control of every aspect of the GEDAE application. It provides a function for reading a launch package and returning an application handle. The application can then be loaded, reset, started and killed using the API. Handles to application parameters and data streams can be obtained from the application handle. Parameter values can be set to control the algorithmic behavior of the graph. And data can be read and written from the application. In addition the command program API allows applications to be remapped at runtime allowing the processors that implement portions of the application to be changed. The COMMAPI also allows multiple GEDAE launch packages to be dynamically connected and disconnected to provide high-level mode control of the running application.



The COMMAPI Provides Control of Running Applications

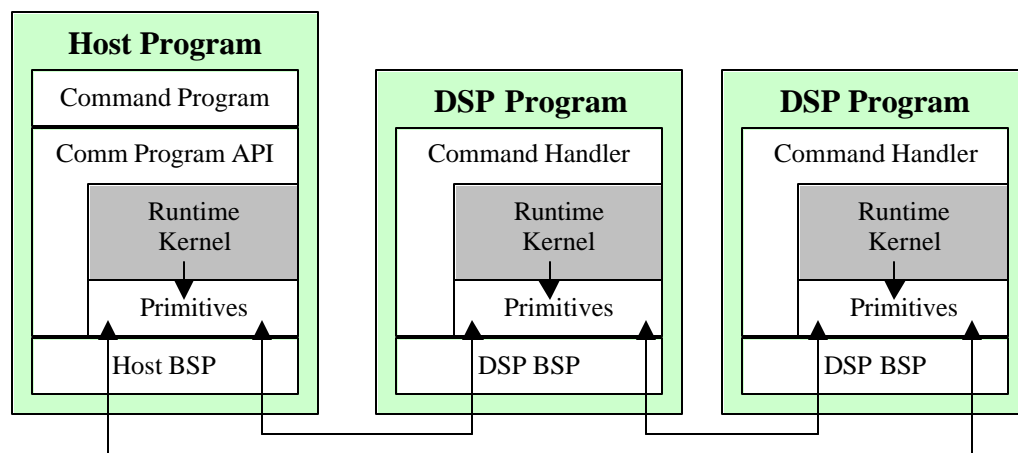
The COMMAPI provides a portable interface to the running GEDAE application since it provides control over the application in terms of the original graph independent of implementation parameters. Thus re-mapping the graph to a different set of processors does not change the command program. In fact as long as the original graph interface to data streams and parameters is maintained even changing the graph will not affect the command program.

The GEDAE development environment is built on the same command program API as user graphs meaning the performance a user observes will be the same whether run from the development environment or user command program. Thus optimizations of the graph observed at development time will be maintained in the delivered application.

Distributed Control of Data Flow – The Runtime Kernel

The GEDAE Runtime Kernel is heart of the GEDAE application providing efficient, scalable distributed control of the data driven application. The data driven design means that each processor in the system will process data as it arrives from other processors in the system so there is no central point of control in the system.

The runtime kernel combines the efficiency of static scheduling and the powerful data-flow control allowed by dynamic scheduling. The kernel executes preplanned static schedules with very little runtime overhead. It implements dynamic data flow using its dynamic scheduler allowing users to build control into their graphs.



The Runtime Kernel provides efficient data driven control

The dynamic schedule is the top-level schedule that is executed by the runtime kernel. A dynamic schedule can consist of one or more static schedules. The order that the next runnable static schedule is executed is determined by dynamic scheduling policy for that particular static schedule. The developer can set a number of parameters that effect how a static schedule is dynamically scheduled including:

- Fixed priority – highest fixed priority schedule runs first
- Queue Policy – can be LIFO or round robin.
- Timeout Policy – set the maximum length of time a schedule can be inactive.
- Ready Policy – Affects when a schedule is ready to fire based on availability of data on the dynamic queues.
- Dynamic Queue Capacity – Set the maximum number of tokens that can be placed in a dynamic queue before an upstream static schedule will block.

The preplanned static schedules are generated under user control at development time so that all of the static scheduling decisions can be made off line. Thus the runtime kernel

for the static scheduler is small and efficient. The developer can control the following aspects of the static scheduling prior to runtime:

- partitioning & mapping – specify on which processor primitive functions will run.
- priority – where there is a choice specify the order of primitive execution.
- granularity – specify how many tokens a primitive operates on in each execution.
- memory mapping – specify in what memory bank each primitive input and output is allocated
- memory packing – the algorithm by which static schedule inputs and outputs are packed in memory as the schedule unfolds.

The control over how a static schedule is created is not part of the runtime kernel and that's its advantage! The static schedule engine efficiently executes the schedule created at development time. At runtime the static schedule executes as a simple loop of the form:

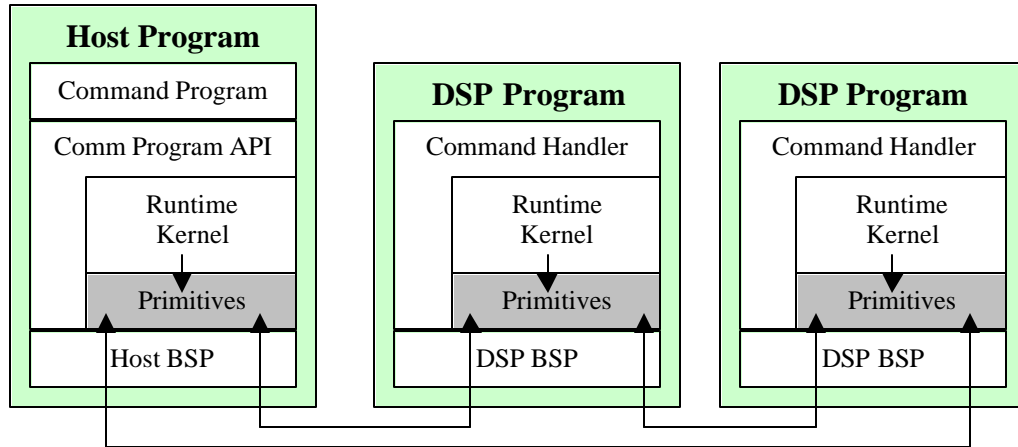
```
runStaticSchedule(Schedule s) {
  foreach closure c in s {
    c->apply(c->state);
  }
}
```

Each closure represents the execution of a primitive. The priority specifications are reflected in the order that the closures appear in the schedule and the granularity, memory mapping and packing specification information are all contained in the closures state vector. The scheduling loop is actually a little more complicated as it must take into account that static schedules can be blocked due to the execution of a closure noting that it is blocked (as when a receive primitive doesn't have data since the sender has not yet fired). Both instrumented and non-instrumented versions of the loop exist to allow trace information to be collected when execution tracing is turned on.

All this control allows the user to tune the application to achieve high performance, low latency, and efficient memory usage. The ability of GEDAE to generate applications as a dynamic schedule controlling static schedules efficiently combines the power of dynamic data flow control with the simplicity and efficiency of static scheduling.

Primitive Function Boxes

Primitive functions perform the actual work of the data-flow-graph. They are the way that a user introduces C code and assembly code that implements an algorithm into an application. An algorithm is specified as a hierarchical data flow-graph with primitives being the lowest level boxes in the graph. Underlying the primitives is C source code and calls to assembly coded routines that implements the primitive's algorithm. While GEDAE provides a growing library currently containing about 4000 standard function primitives a user can develop custom primitives to handle specific needs.



Primitive functions drive the application dataflow

New primitives are easy to write. GEDAE's primitive syntax makes easy primitives easy. User and standard primitives use generic vector processing library built on top of optimized libraries supplied by the vendor providing both efficiency and portability.

```

Name: add
Type: static
Input: {
    stream float a;
    stream float b;
}
Output: {
    inplace stream float out=a;
}
Include: {
#include <e_vadd.h>
}
Apply: {
    e_vadd(a,1,b,1,a,1,size(a)); /* vector add */
}
    
```

The Name, Input, and Output sections describe the interface to the primitive

add	
a	out
b	

Using standard vector calls makes user primitives portable and fast

New primitives are easy to write to extend GEDAE

But GEDAE also provides the ability to implement complex data-flow control. Features include:

- static data rate modification – a primitive can declare that for every n tokens received on an input it will produce m tokens on its output
- overlapped data processing – a data stream can be overlapped between firings allowing windowed filtering functions to be easily implemented.
- cyclic functions – data flow can change from firing to firing in a static cyclic pattern that can be efficiently scheduled. This allows demultiplexing primitives to be implemented with low latency.
- dynamic determination of data flow - GEDAE provides a powerful ability to specify dynamic data-flow in a primitive. Primitives can declare input or output ports to be dynamic and determine at execution time how much data is produced or consumed. Dynamic primitives allow control to be built into a graph.
- encoded data streams – data streams that change rarely can be more efficiently implemented as run-length-encoded data streams. Connecting such streams to parameters efficiently promotes the parameter to a stream.

GEDAE also allows primitive to maintain local state information private to the instance of the primitive on the graph. Such state information is useful in implementing filtering algorithms or in maintaining handles to operating system resources such as file descriptors.

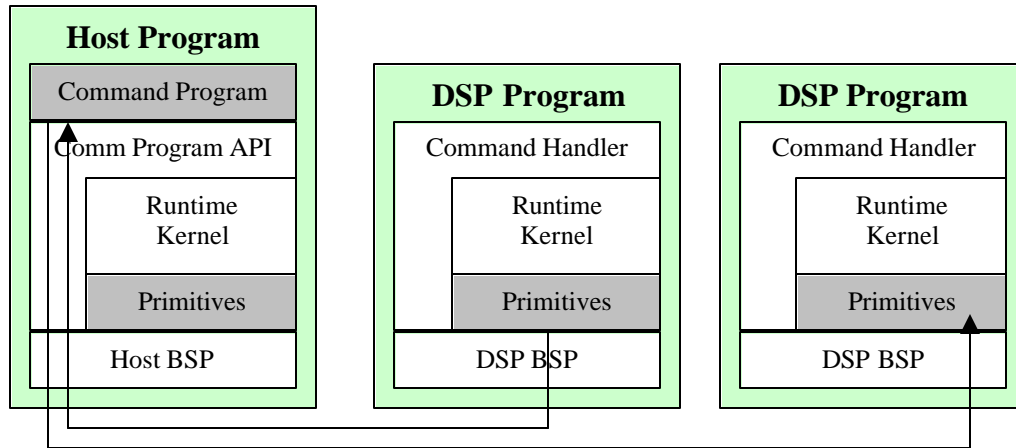
In addition to the primitives added to the graph by the user GEDAE automatically adds primitives to the schedules to perform various functions:

- enqueue/dequeue – these primitives implement the communication to and from the command program data ports.
- send/receive – these primitives implement the communication between different processors.
- dyndeq/dyndenq - these primitives allow primitives with dynamic ports to be connected directly to primitives with static ports
- subschedule – this primitive allows portions of a graph to be subscheduled at a finer granularity within a large static schedule. Subschedules reduce latency and allow large data sets to be “strip-mined” to gain higher efficiency in cached memory systems.
- copy – the copy primitive moves data within the schedule as needed to handle data-flow requirements.
- sync – this primitive allows parameters to be synchronized with the data-flow.

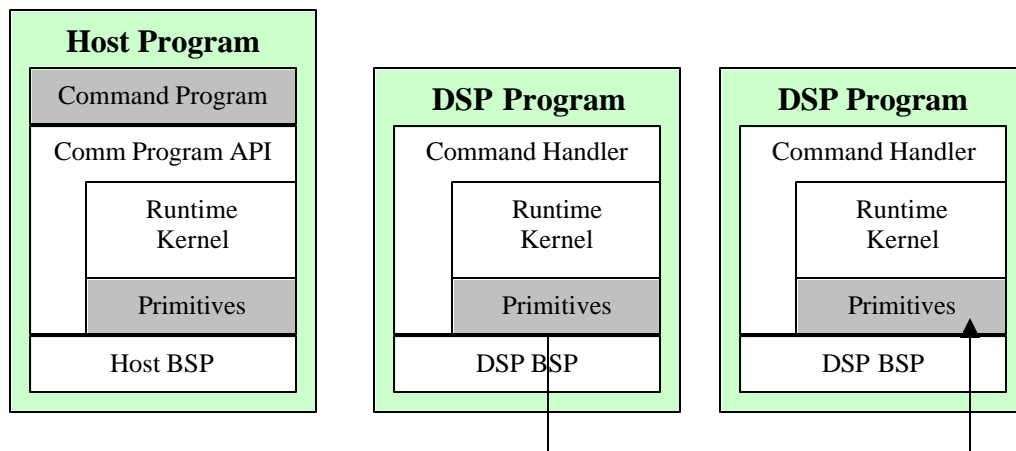
Adding all functionality to a graph as primitives makes GEDAE applications easy to understand. The static schedule display, that presents the predetermined order of primitive firing, shows both user and auto-generated primitives. Also the Trace Table, which presents function firing vs. time, shows the execution of both types of primitives.

Communication between Command Program and Application – Host to Embedded Communication

The Command Program has efficient data driven control of graph execution through the ability to read and write data directly to the executing graph. Generic enqueue/dequeue primitives built on the thin BSP layer provide an efficient interface between command program data and the rest of the GEDAE graph.



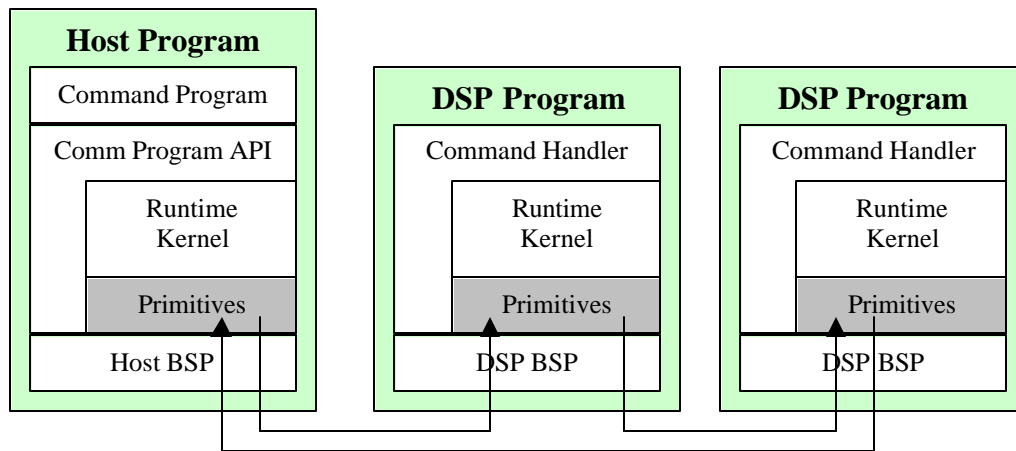
Command Program sends and receives data from application...



... and can reconfigure the application at runtime.

GEDAE allows communication to be reconfigured at runtime which is useful in implementing major mode changes in a graphs execution. Applications that start off by communicating with the host can be reconfigured so that they communicate directly and efficiently with each other without host involvement. The host can still modify parameters and monitor the application or it can exit leaving the DPS Program to continue on its own.

Communicating Between Application Partitions Efficiently – Embedded to Embedded Communication



Inter-processor communication approaches hand coded performance.

GEDAE's data driven design allow generic send/receive primitives to perform efficient point to point communication through the thin BSP layer. The BSP routinely supplies communication performance comparable to hand coded performance.

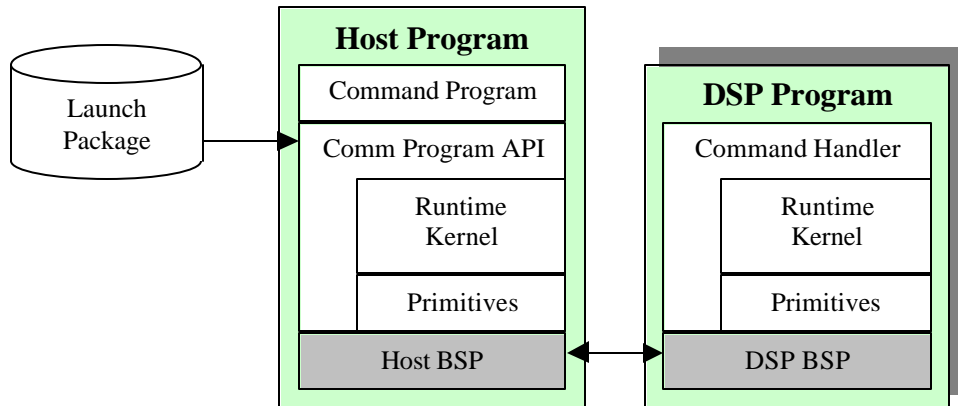
On different hardware GEDAE supports different mechanisms, but typically GEDAE supports socket based communication, shared memory communication and DMA based communication. The developer can select the best communication mechanism for a given communication channel and can parameterize the channel in terms of the size of the data buffer and number of buffers. This ability to control the communication allows the developer to trade off the speed of the communication against its memory requirements.

For processors that allow parallel DMA data transfers, the GEDAE Direct Schedule Access Mechanism (DSA) allows multibuffered IO to be performed in parallel with processing achieving the highest possible communication efficiency. DSA performs its communication directly from the buffer that last primitive filled into the buffer that the next primitive will process from. So no copies of data are required and all transfer of data is done in the background.

The GEDAE development environment automatically adds the send/rcv primitives needed based on the user partitioning of the graph. And it selects the right type of send/rcv primitive for the data being communicated. Different types of primitives are available for normal streams, encoded streams and streams whose tokens are complicated data structures. Special receive primitives are also available to make communication to a dynamic data stream more efficient.

The Board Support Package

GEDAE's Board Support Package (BSP) provides a thin layer over the hardware specific BSP giving efficient, portable access to the hardware. It also provides a thin layer over the hardware specific vector library allowing function primitives to get portable access to efficient vector library calls.



The BSP Provides Efficient Portable Access to Hardware

The BSP provides direct access to multiple vendor communication mechanisms allowing a developer to select the most efficient mechanism for the type of communication required. It provides communication and other resource allocation and deallocation allowing the command program to dynamically reconfigure the application at runtime.

The GEDAE BSP Development Kit allows developers to create a BSP for their hardware giving the developer the ability to maintain the BSP and maximize its performance. GEDAE already has BSP's that support the following hardware and operating systems.

Vendor	DSP Processor	OS Version	Host Processor/OS
CSPI	PPC	VXWorks 1.5 Myrinet 3.1.2 MPI_PRO 2.0.7 ISSPL 2.24	Sparc/Solaris
Ixthos	Sharc	IXZtools 5.2 ADI DSP compiler 3.3 IXLibs21k 2.0	Sparc/Solaris, PC/NT
Mercury	PPC	Mcos 4.4.3	Sparc/Solaris, PPC/Mcos
Sky	PPC, Altivec	Skyvec 5.4.0.1	Sparc/Solaris
Sun	Sparc	Solaris 2.6	Sparc/Solaris
PC	PC	Windows NT 4.0	PC

Refer to the GEDAE™ Tech Doc #1 for more information.