XIE Sample Implementation Architecture

Authors

Robert NC Shelley

Dean A. Verheiden

J. Ben Fahy

Sample Implementation Team

AGE Logic, Inc.

Additional Contributors

Larry Hare

AGE Logic, Inc.

Comments concerning the SI architecture should be submitted by electronic mail to:

imagework@x.org

Acknowledgments

XIE has evolved from the original specifications and sample implementation developed by Digital Equipment Corporation and contributed to the X Consortium.

We wish to thank Digital Equipment Corporation and Hewlett-Packard Co. for contributing their pipeline implementations to the X Consortium. The SI architecture has benefited from these efforts.

Table of Contents

1. Introduction

The X windows Image Extension (XIE) project was initiated in 1988 by Joe Mauro of Digital Equipment Corporation, in order to meet the needs of those wanting improved imaging facilities within X. XIE is targeted at the problem of image display, and rendering any digital image on any X-capable hardware. XIE is not intended to be a general purpose image processing engine, although it does provide a rich environment for performing simple image enhancement operations and format conversions. XIE also permits the support of image-specific hardware in the server, including but not limited to decompression, scaling, rotation and convolution accelerators.

The X Consortium accepted XIE into Technical Review in March of 1991. After over a year and a half of debate and analysis by a number of interested Consortium members, a revised Version 4.12 of the XIE protocol was passed out of Technical Review and into Public Review. The Public Review period is scheduled to expire April 23rd, 1993. It is the goal of the Consortium that XIE will be made widely available shortly after acceptance as an official standard, and therefore AGE Logic, Inc. has been commissioned to provide a Sample Implementation (SI) of XIE, to be distributed as part of the December 1993 X windows R6 release. This document describes AGE Logic's initial design of the XIE SI architecture.

The audience of this document is anticipated to be X Consortium members who have a working understanding of XIE, either through participating in its design within the Consortium, or by having read the latest version of the XIE protocol, by Shelley et al., published in Special Issue C of The X Resource journal. An overview of XIE by Fahy and Shelley is also provided in the same issue. Other references of interest would include <u>The X Window System Server</u> by Israel and Fortune, and <u>X Window System</u> by Scheifler and Gettys.

1.1. Task Definition

XIE is defined by the protocol document referenced in the previous section. XIE *requests* are channeled through the Core X dispatch loop to the dispatchers (swapped and unswapped) for the XIE SI. XIE-specific *events* and *errors* are originated in SI and routed back through Core X to the waiting client.

XIE defines six new resources: ColorLists, LUTs, Photoflos, Photomaps, Photospaces, and ROIs. Each of these resources must be recorded with Core X, so that at client shutdown the resources may be freed reliably.

In the XIE computational model, a client provides a blueprint for a computational engine, a *photoflo*, and passes it to the server for execution The blueprint is a directed acyclic graph (*DAG*), which shows what the components (*elements*) of the engine are, and how data flows between them. *Import* elements receive data from the client, Core X, or from resources. Processing elements receive data from import elements or upstream processing elements, perform a specified operation on the data, and pass the data on downstream. *Export* elements define the exit point of data from the photoflo. Data from Import or Processing elements may be routed to the client, Core X, or an XIE resource simply by adding an Export connection after the element of interest.

Processing, Import, and Export elements are defined at two levels. For example, a *Geometry* element states that incoming image data is to be re-sampled to produce a new output image. The specific sampling mechanism is selected by using a set of *technique* parameters in the element specification. Thus, technique parameters give fine-grain information on how a course-grain operation (such as geometric transformation) is to be performed.

To summarize the tasks which the SI must perform, it must:

1. Integrate into Core X
2. Receive and correctly interpret XIE-specific requests
3. Manage XIE-specific resources
4. Build computational engines (photoflos) according to the blueprints (DAGs) provided
5. Execute the photoflos
6. Return error and event information as required.

Building and executing the photoflos necessitates:

1. Parsing the DAG information
2. Performing semantic analysis
3. Matching sub-tasks (defined by elements, etc.) to machine-specific code
4. Scheduling execution
5. Managing data flow.
6. Handling special conditions, including shutdown and return of control.

Note that the SI must be able to support multiple photoflos executing simultaneously, and thus all photoflo and element code must be reentrant, and data structures must not be written so as to preclude reentrancy.

1.2. Design Goals

The MIT X Consortium, in soliciting for bids on the SI project, has previously stated the following design requirements:

1. The SI must be easily ported to many different platforms.
2. The SI must be modular, to support easy customization.
3. Performance should be commercial quality.
4. Performance should not be biased to a particular machine architecture (e.g., big endian machines should have performance comparable with little endian machines of the same approximate raw speed).
5. The SI should accommodate servers with limited memory.

An additional, implied requirement is:

6. Design of the SI should be minimal risk.

That is, it must be acknowledged that while there are any number of ways in which photoflo optimizers and schedulers could be implemented, the overriding goal must be to ensure that an SI is available in time for the R6 release of X windows.

1.3. Document Organization

This chapter has provided a brief introduction to the topic of designing a sample implementation of the X image extension. The next chapter provides an overview of the architecture as the design currently stands. It starts by looking at XIE from a very high conceptual level and working its way down examining a few of the SI's major building blocks.

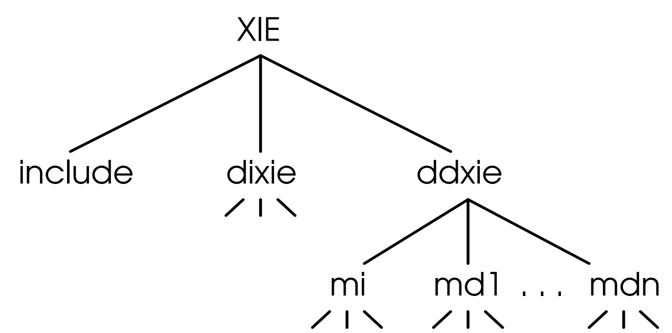
The next three chapters discuss in more detail: extension initialization, memory management, and the dispatching of requests through Core X. They are followed by a brief description of a some high-level data representations, followed by a detailed exposition of critical data structures. These chapters serve as references for the chapters that follow.

The eighth chapter discusses the implementation ramifications of XIE requests, up to the topic of photoflo management in the server. That issue is covered in detail in chapters nine and ten, which are split across device-independent / device-dependent lines. The final chapter discusses elements; their data structures and interfaces.

2. Architecture Overview

Core X is divided into device-independent (dix) and device-dependent (ddx) layers. The dix layer contains code and data structures which should not need to be altered in porting X from one machine to another. The ddx code is where device-specific information, such as drivers for special hardware, assembly routines for optimized graphics, etc., are placed. The dix layer accesses ddx code by calling ddx procedures indirectly, through a list of well-defined entry points stored in dix resources.

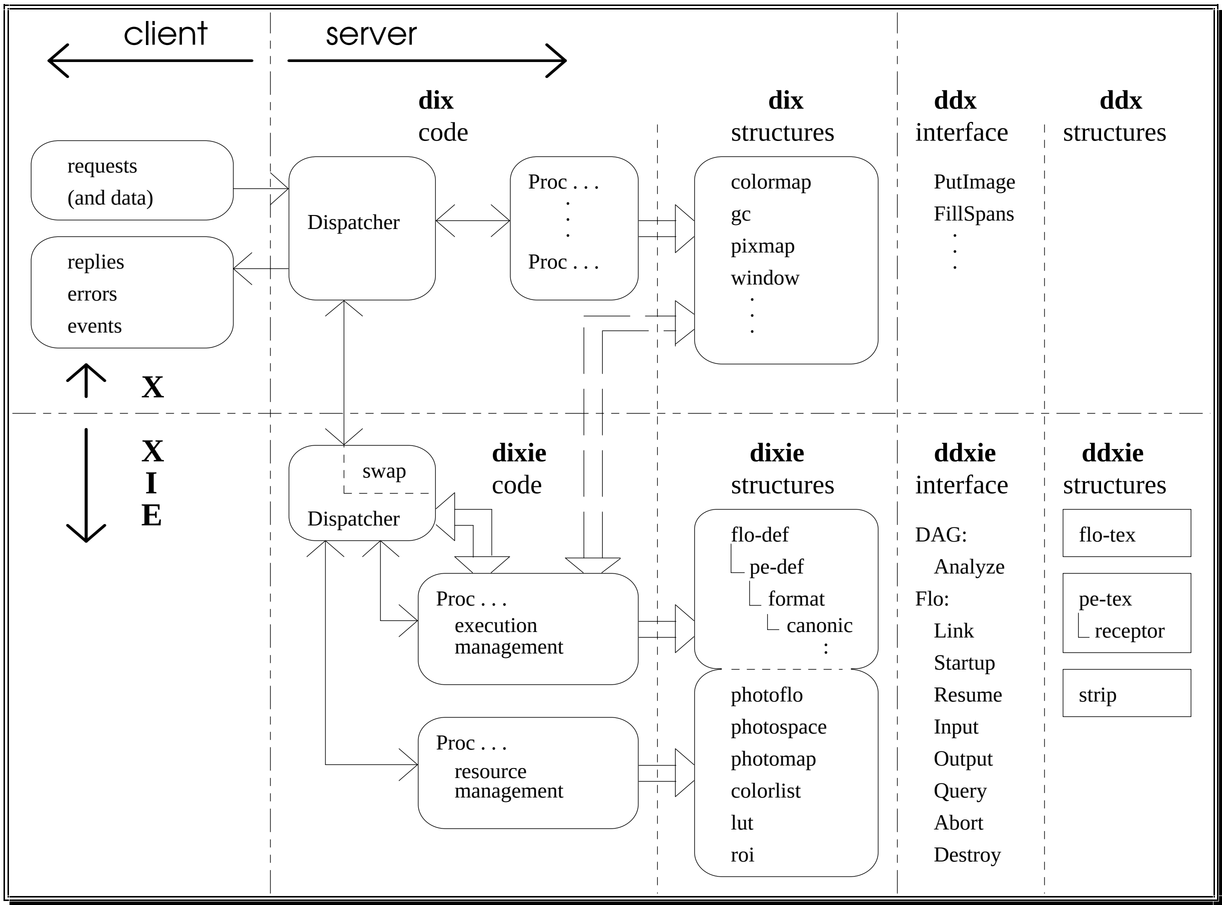
The design of XIE is intentionally modeled so as to follow the same basic form. XIE is divided into a device-independent layer called *dixie*, and a device-dependent layer called *ddxie*. A set of dixie data structures is defined, which includes a list of entry points to ddxie routines. In the SI, this list will be filled exclusively with machine-independent XIE code. However, in the interest of portability and optimization, care has been taken to make it easy to override the generic code with machine-specific routines. It is anticipated that server implementors porting to new hardware will create new ddxie subdirectories which are machine dependent, as is depicted below.



XIE directory layout

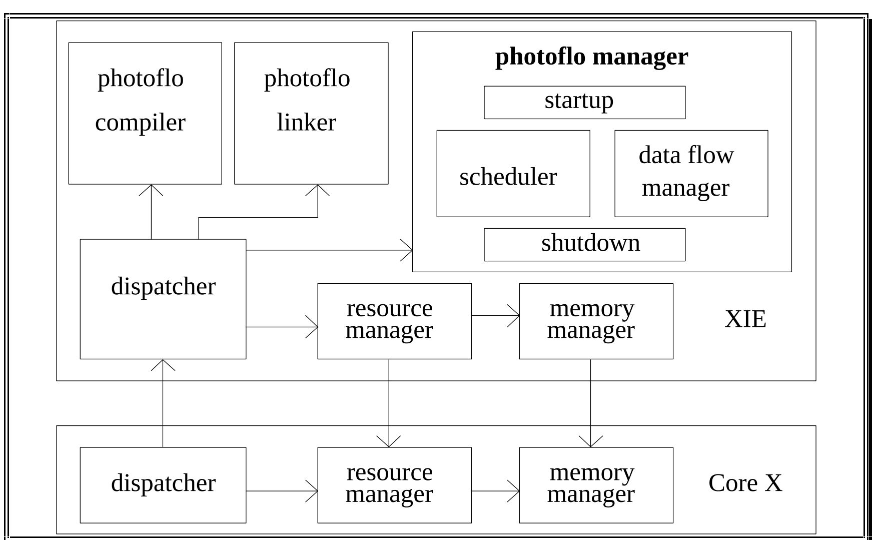
2.1. X/XIE Interfacing

The figure on the following page illustrates the interface between the new XIE architecture and the existing Core X server design. All routing of data in or out of XIE must go through Core X. XIE requests are received by the Core X dispatcher, recognized as belonging to XIE, and the appropriate procedure to receive the request is called. Similarly, errors and events generated within XIE are routed back out using standard Core X mechanisms. Image, LUT, and ROI data is passed from the client to XIE via a PutClientData request, and thus follows the same route as normally dispatched data. Image, LUT and ROI data to be returned to the client is sent via Reply packets, using the conventional Core X technique. Finally, XIE moves data between an XIE photoflo and Core X resources by making use of the list of Core X entry points described above.



2.2. XIE Components

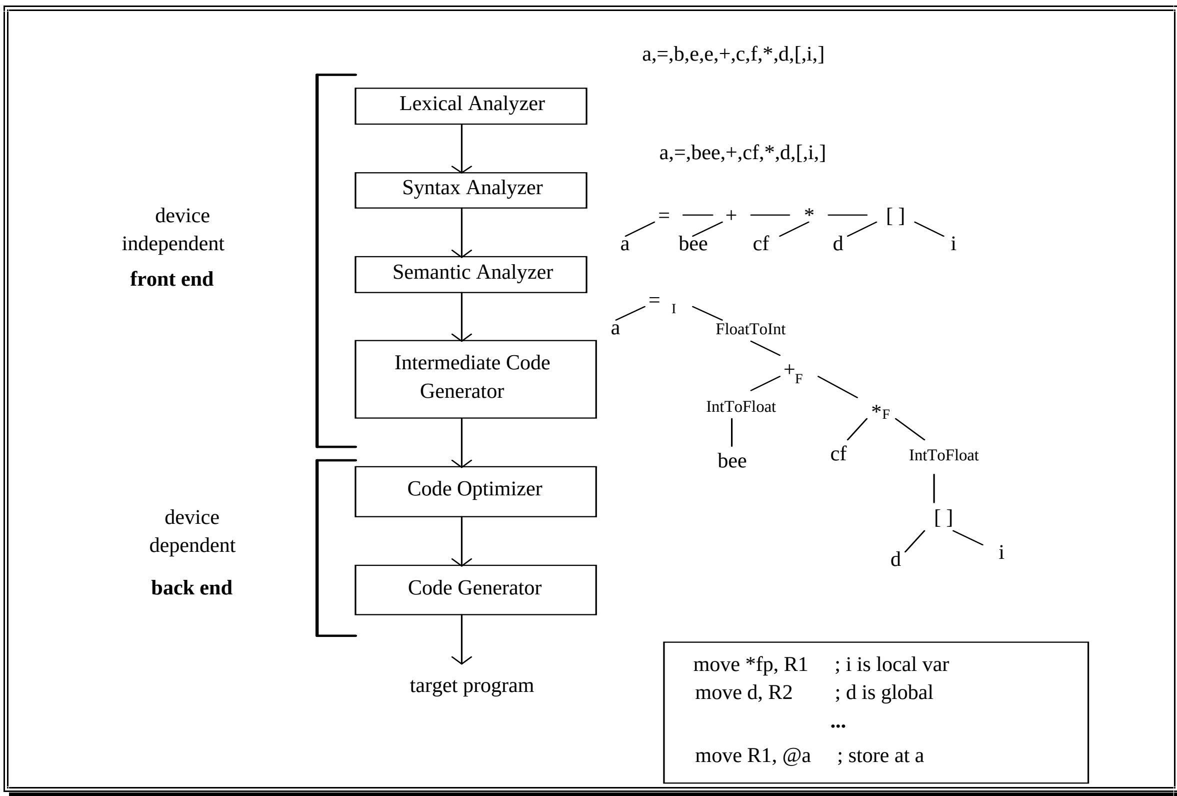
At the highest level, XIE may be viewed as containing the components depicted in the figure below (it also contains other components, not shown). Core X provides a resource manager in the dix layer and a memory manager in ddx. XIE will provide its own resource and memory managers layered on top of the Core X facilities. XIE also has its own dispatcher. Routines within the XIE dispatcher are called when the X dispatcher receives a request which it recognizes as belonging to the imaging extension.



When the request contains the definition of a photoflo, this definition must be translated into the appropriate code and data structures required to build and execute the flo. For this reason, XIE may be viewed as containing a Photoflo Compiler, which performs this task. The output of the compiler is a set of data structures which describe how the photoflo will be executed, given the existing available code. Initialization of static variables and resolution of entry points identified by compilation is the responsibility of the Photoflo compiler. Execution is conducted by a Photoflo Manager, which consists of a Scheduler, Dataflo Manager, and other components. Client data is input through the XIE dispatcher and given to the Photoflo Manager, which deposits the data in with the Dataflo Manager and then wakes up the Scheduler.

2.2.1. Photoflo Compiler

The sub-structure of the Photoflo Compiler may be derived by comparing the task of compiling photoflos to classical compiler problems. The figure below shows the classical model of a compiler which is producing code for the expression **a = bee + cf*d[i]**, where **cf** is floating point and **a, bee**, and **d** are integers.



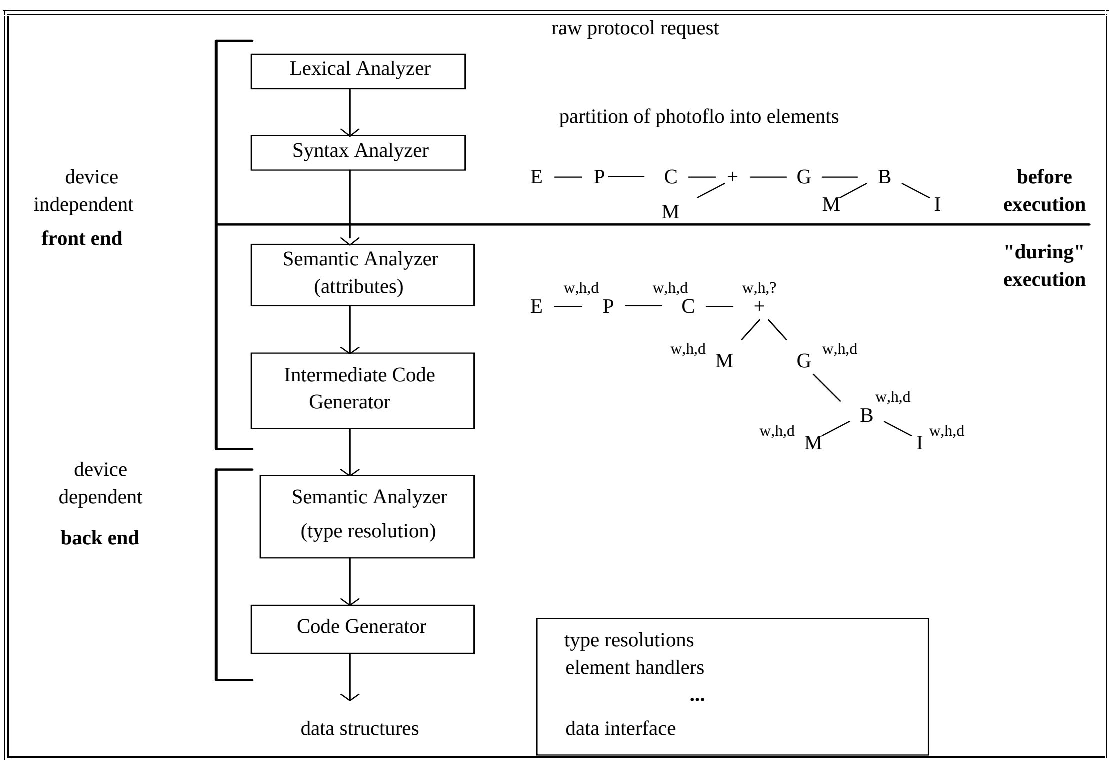
The input to the compiler is simply a stream of bytes, that is, **a** followed by **=** followed by **b**, followed by **e**, etc., with no meaning or structure yet attached to the stream. *Lexical analysis* groups the characters together into the distinct tokens, **a**, **bee**, **+** , **cf**, etc. *Syntax analysis* is used to resolve the order in which fundamental operations are carried out, which is represented as a directed graph. *Semantic analysis* adds type and other attribute information to the problem representation. Note that required type conversions and operator classes have been made explicit in the output of the Semantic Analyzer. *Intermediate Code Generation* converts the DAG representation into a device independent pseudo-language (output not shown). The first four phases of the compiler are often grouped together and referred to collectively as the compiler *front end*, which is the device independent part of the compiler.

The *back end* of a compiler is the device dependent part, which consists of the optimizer and code generator. Most compilers incorporate both device independent and device dependent optimizations. The code generator translates from the intermediate language representation to the language of the target machine.

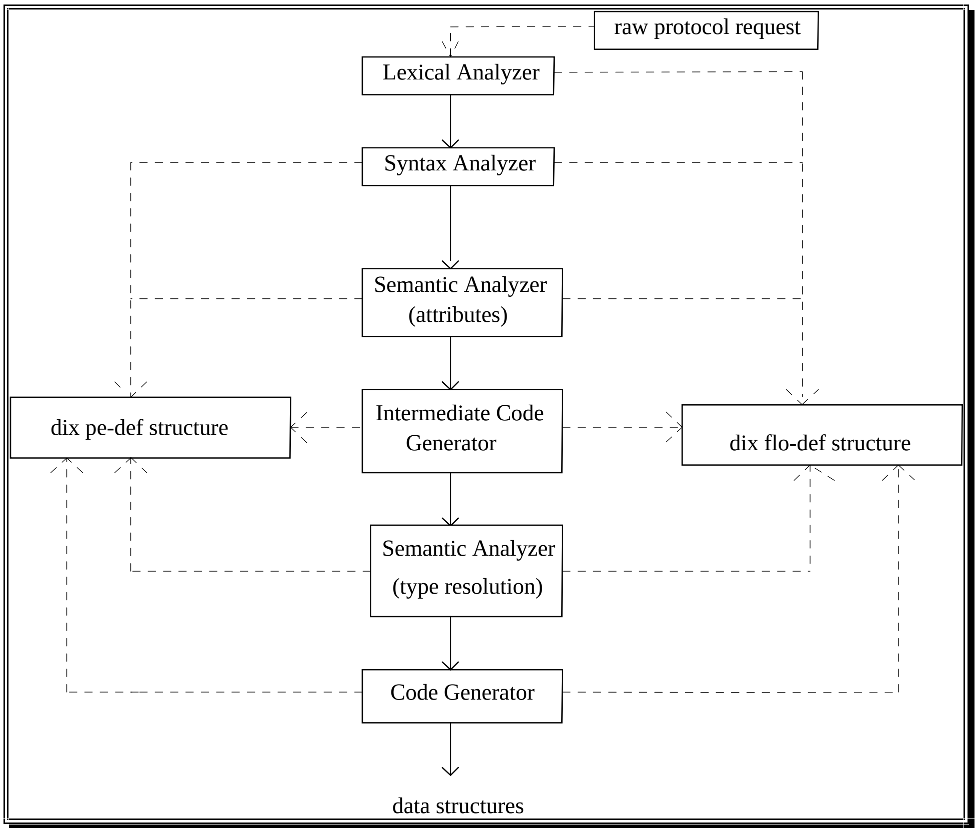
In attempting to apply the standard compiler model to the problem of photoflo compilation, a number of fundamental differences in the two environments must be recognized. First, it should be noted that there is usually a significant delay between when a conventional compiler translates source code into object code and when the final executable is eventually run. This means that the time constraints on the compiler are not necessarily severe, allowing for more extensive optimization. In contrast, in XIE, it will *always* be the case in compiling photoflos that significant parts of the compilation time will occur just before execution, and add to the perceived execution time. This is because the XIE protocol expressly forbids resolving the attributes of input data before the ExecutePhotoflo (or equivalent) request is given. This puts a hold on semantic analysis, constraining it to wait until just before execution. Since code generation and optimization follow the semantic analysis phase of compilation, these phases are also constrained, and the time to carry them out should be kept tightly restricted. This inhibits exhaustive optimization, and encourages very simple code generation strategies.

Another important environmental difference which affects the application of the compiler model is that most conventional languages expose the canonical data types of the processor fairly conveniently. The underlying hardware model assumes registers of a few fixed sizes, whereas in XIE, the "register size" of the server is hidden completely. Canonical data types, such as 1, 8, and 16 bits might be chosen by the server implementor in order to reduce the number of cases for which processing elements must be written. However, this may vary from server to server, and thus the SI code should not make the decision that it already knows all of the canonical data types. Although the SI will support 1, 8, 16 and 32 bit data, it is also a design goal that the SI be easily extensible if a vendor also wants to support 4 bit data. This implies that the type resolution portion of the semantic analyzer must be moved out of the device independent front end, and into the device dependent back end.

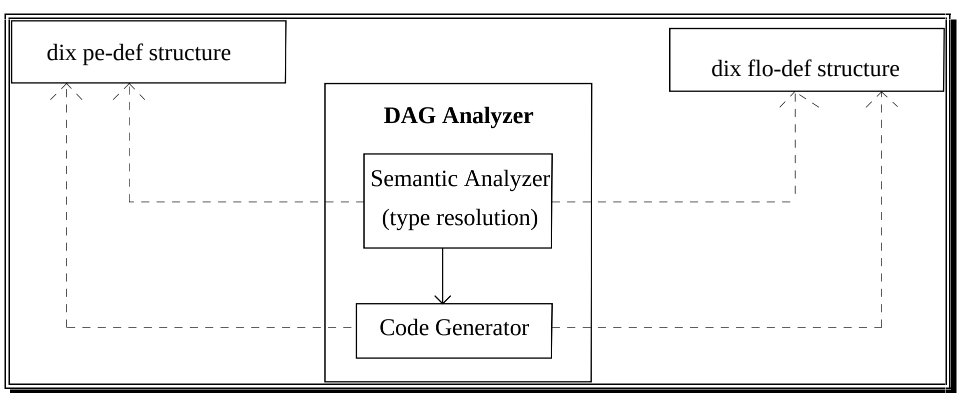
The figure below shows the basic compiler model modified to accommodate the XIE computational model. Only lexical and syntactic analysis may occur prior to receiving the equivalent of an ExecutePhotoflo request. Lexical analysis in the context of XIE refers to parsing the raw data stream and breaking it into the element definitions, etc. Syntactic analysis refers to analyzing the DAG to make sure it is legal (e.g. contains no cycles). As the DAG is analyzed, the dixie data structures which describe it are filled in.



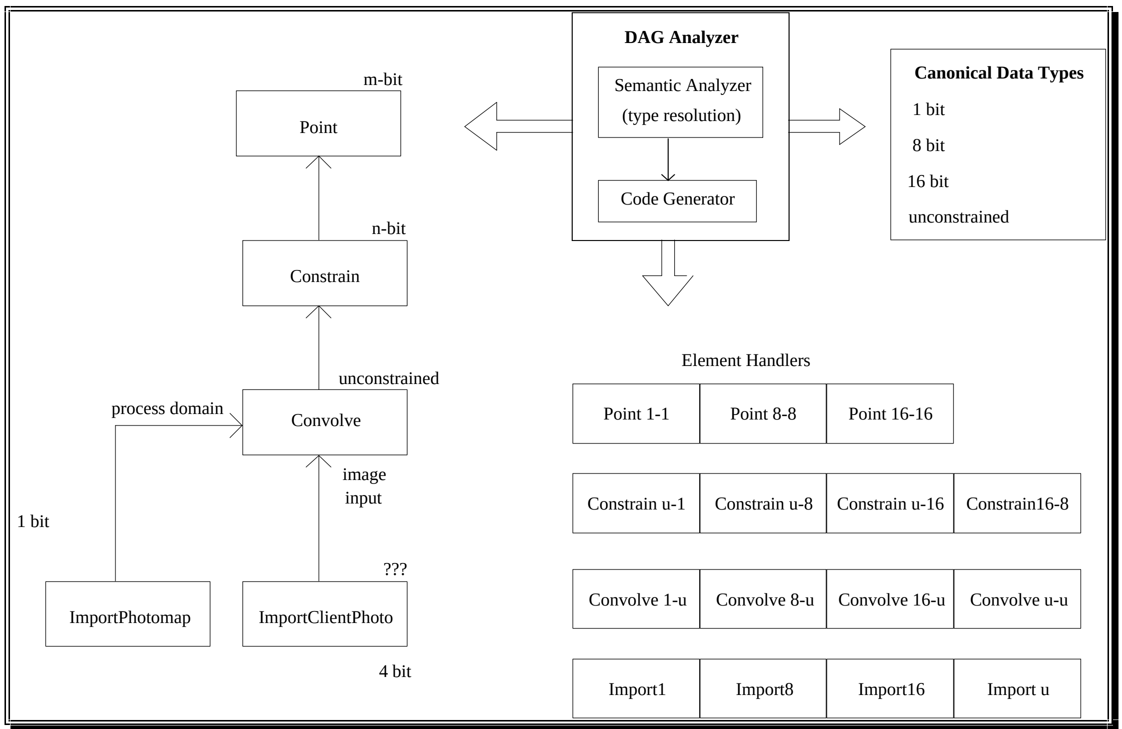
The semantic analyzer is split into device independent and device dependent portions. Propagation of attributes from import elements to downstream elements occurs in the device independent part of the semantic analyzer, whereas type resolution requires device dependent information. Both parts of the semantic analyzer fill in structures in the dix layer of XIE (see figure, following page). The difference is that the attribute analyzer does not use any device independent information to produce its data, whereas the type resolution code does.



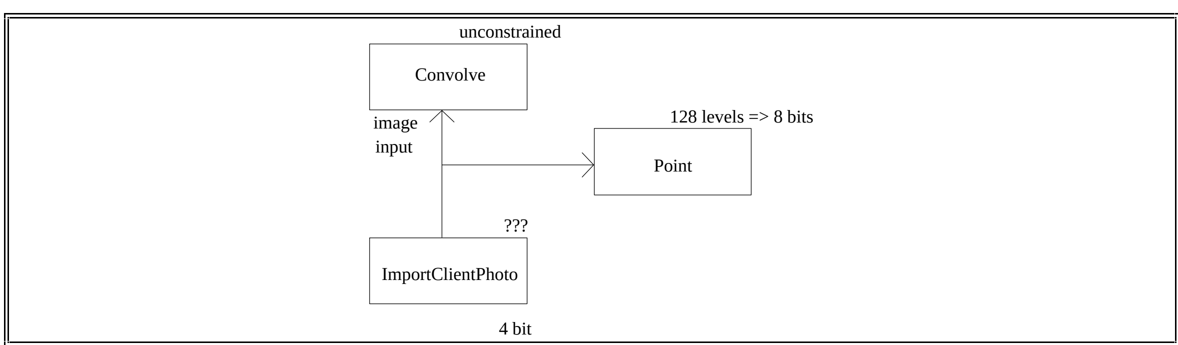
The code generator has the task of accepting the device-independent photoflo representation as input, and filling in the device-dependent data structures which implement the photoflo. Because code generation may be closely coupled with type resolution, the SI groups these two operations into a DAG Analyzer, as is depicted in the figure below. The SI will include a default machine independent DAG Analyzer which handles type resolution by a simple type upgrading scheme, and which generates code by mapping directly from elements defined in the DAG to mi element *handlers*. The design accommodates substitution of a device dependent DAG Analyzer, however, which can choose to perform special optimizations, support non-SI canonical data types, and employ a machine-dependent element interface.



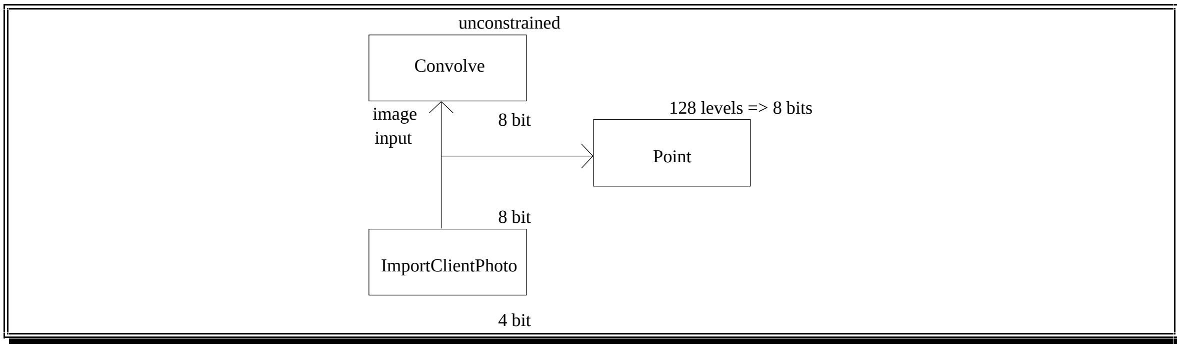
The figure below illustrates the fundamental task of the DAG Analyzer. A Photoflo to be compiled is shown at the left, in which a one-bit image is imported as a process domain, a 4 bit image is imported and convolved, the output is constrained, and a lookup table transformation is applied using Point. The XIE server implements Point, Convolve, Constrain, ..., operations on various canonical data types. By definition, element handlers only accept canonical data as input, and only produce it as output. If the routines to perform a convolution can only accept 1,8,16, or unconstrained data, and the routines to import client image data can also produce any of these types, it is necessary to negotiate on which type should be used between connected elements.



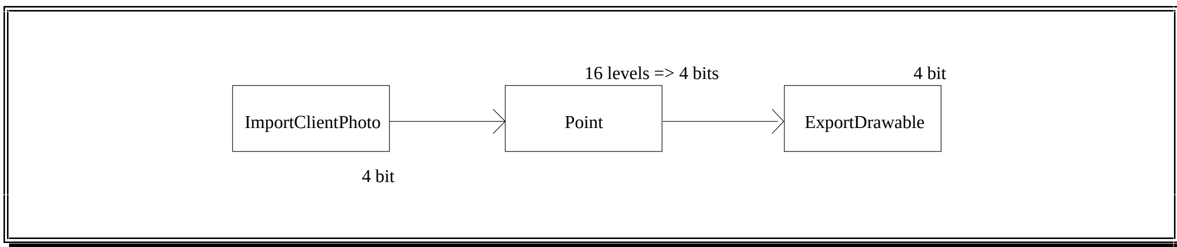
Consider the photoflo depicted below, which has 4 bit image data being fed to both Convolve and Point. The user has specified in the protocol that the Point element is to produce data with 128 levels, which requires at least 7 bits of storage. This suggests that the output of the Point element should produce 8 bit data, which is the closest canonical form with adequate resolution.



Using the same reasoning, that is, that elements should produce as output the smallest-resolution canonical form which is adequate, ImportClientPhoto should produce 8 bit data. This would fix the type at the input of the Convolve element, implying that it is necessary to choose a Convolve handler which accepts 8 bit data.

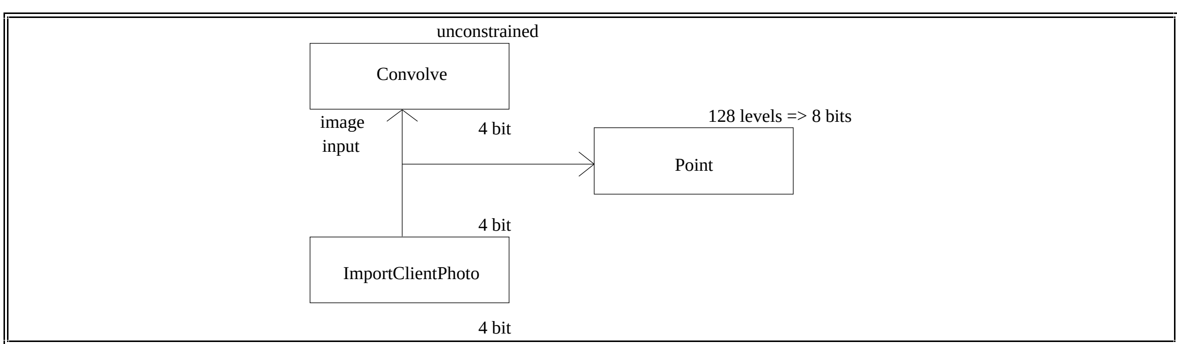


Let us now suppose that it is desirable that server implementors be allowed to introduce new canonical types, such as four bit data. This would be quite natural on servers which only had four bit deep displays, and wanted to insure good performance on operations like:

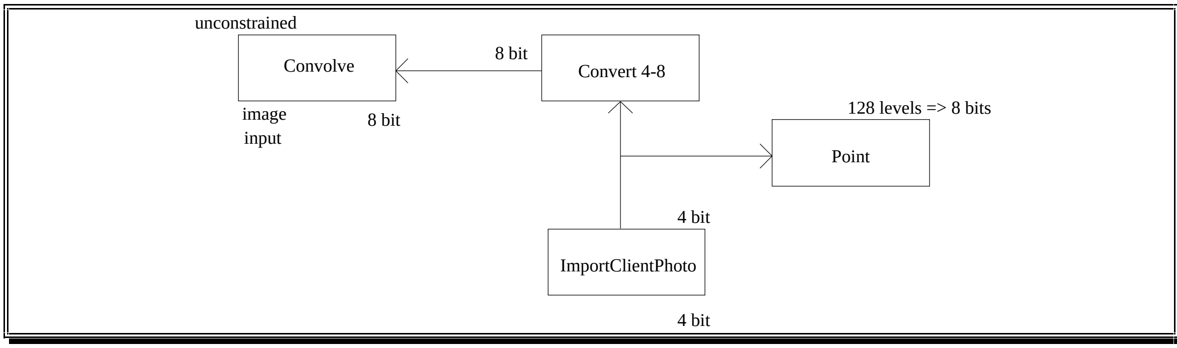


Rather than have to convert the 4 bit data to 8 bits, do an 8 bit Point and then convert back to 4 bits in ExportDrawable, it might be more efficient to keep the data as 4 bits throughout. This implies that the implementor might want to define an ImportClientPhoto handler which could produce 4 bits output, a 4->4 Point handler, and an ExportDrawable handler which accepts 4 bits. Let us suppose this is permissible.

Returning to the case of the photoflo containing a Convolve element above, reapplying the heuristic that the output of an element should be in the smallest available canonical form which has adequate resolution, we reach the situation:



The problem is, we are now requiring the presence of handlers for Convolve and Point which can accept 4 bit data and produce unconstrained and 8 bit data, respectively, and these may not exist. It might be reasonable to require a 4->8 Point handler, since that may have general utility, but since Convolve is fundamentally an unconstrained operation, it seems overly harsh to force the server implementor to go through and alter all element handlers like Convolve to include conversions of data types which are not interesting. An alternative is to provide mechanism in the dixie structures for a DAG analyzer to be able to insert "pseudo-elements" which do the conversion or perform analogous tasks:

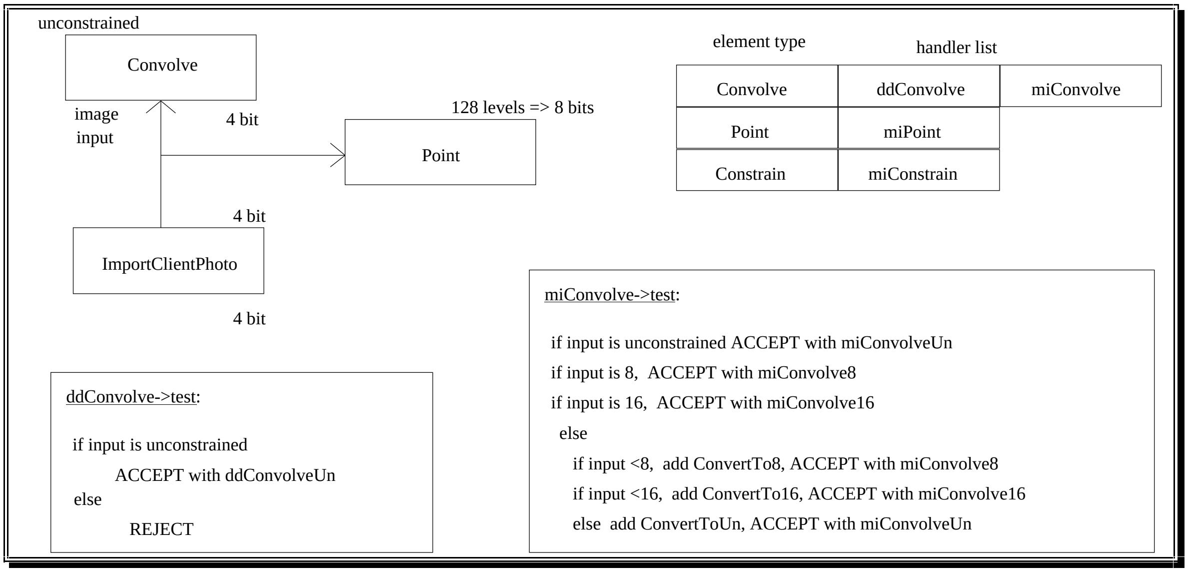


Thus, the SI data structures allow the DAG analyzer to produce a new, optimized DAG with a different number of "elements" than the original flo. The ddxie layer is allowed to define *pseudo-elements* which may perform type conversions, or combine two or more elements into a single "super-element," or perform other functions (for example, converting a bitonal image into a list of run-lengths, if it is to be used as a Process Domain). This provides a great deal of flexibility for future enhancement and optimization.

The SI DAG Analyzer will employ the following general method:

1. For each element defined by the protocol, and each implementation-specific pseudo-element, there must be written at least one handler which is capable of doing processing for that element.
2. All handlers will output a type that is the smallest acceptable canonical type for the input data and the particular operation.
3. Prior to trying to find a handler for a particular element, the DAG Analyzer will propagate the output types of all of its source elements to its input data receptors.
4. Each handler must provide a test entry point, which the DAG Analyzer can call providing the photoflo and tag number of the element to be "compiled".
5. The handler's test function will examine the element parameters and the requested input types and return ACCEPT if it agrees to handle the element, and REJECT otherwise.
6. A handler which agrees to accept an element may change the input type coming from any source element by indicating that a Convert pseudo-element must be inserted between itself and the source.
7. If multiple downstream elements require the same conversion, they will share output from the same converter.

The next figure provides a simplified diagram of relevant data structures. The DAG analyzer is given a photoflo, which is shown at left. It orders the elements by their distance from the nearest Import element, and resolves types in the order of increasing distance. The upper right hand corner of the figure shows a list of handlers associated with each element type. By the time the DAG analyzer gets to the Convolve element, ImportClientPhoto has already been resolved to produce four bit output, and that "suggested input type" has been recorded in the input receptor of the Convolve element.



The DAG analyzer looks at its list of handlers which are of type Convolve, and selects the first member of that list, ddConvolve. This handler supports floating point hardware, and as written, does not insert Conversion pseudo-elements to prepare its input data. Therefore it rejects the element, and the DAG analyzer goes on to the next handler in the list for Convolve.

All elements will contain as the last handler in their list a machine independent handler which can accept any canonical data type. If the data type matches an explicitly supported type, the handler loads the entry point for the specific function and leaves the requested types unchanged. If the data type is not explicitly supported, the handler inserts a Conversion element into the photoflo (these are pre-resolved, so the DAG analyzer doesn’t have to back up to find a handler for it) and accepts with the nearest sufficient canonical type which it supports.

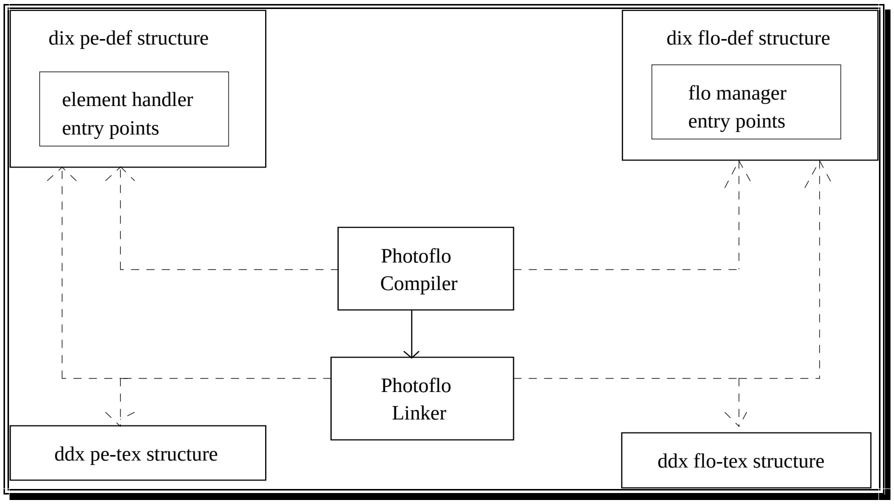
The SI will provide mi handlers which can accept any element specified in the protocol. Note however that the SI may provide multiple mi handlers for a single element, which may be differentiated by which techniques they support, etc.

The method described above solves the code generation problem by effectively labeling each element in the photoflo with a handler. Discussion of the data flow between elements and scheduling of processing is deferred to the section on the Photoflo Runtime Manager.

2.2.2. The Photoflo Linker

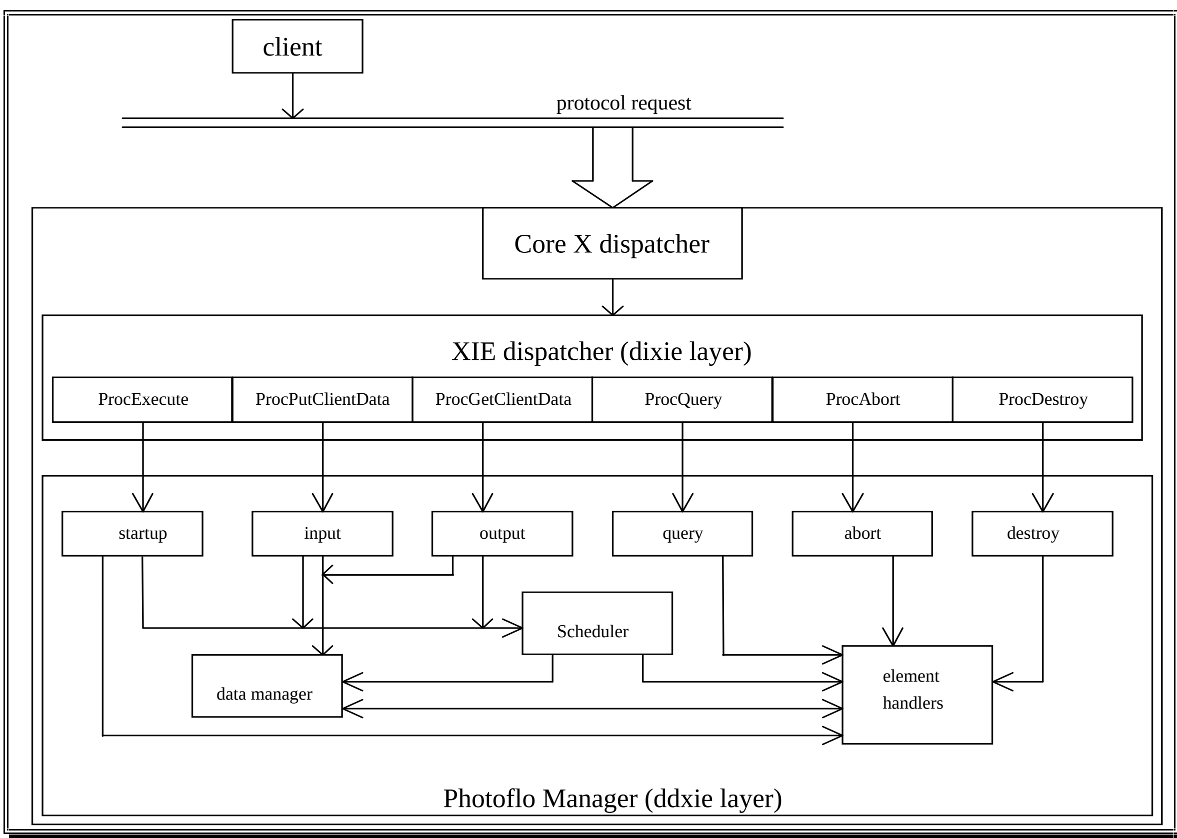
After the Photoflo Compiler has resolved data types and handlers for each element, the Photoflo Linker is called to perform one-time ("static") initialization of the photoflo. It is at this point that the first device-dependent structures are created, which provide an execution context for the photoflo and its elements. Entry points into the photoflo manager are loaded in the dixie flo-def structure, along with entry points to the flo manager’s sub-components, the scheduler and data manager.

Both the linker and the DAG analyzer portion of the compiler can be skipped if the photoflo has been executed previously. This is because the dixie structures retain the one-time initialization state even after shutdown. The attribute checking portion of the Semantic Analyzer must (by directive of the protocol) be re-run at each invocation, but if it discovers that no attributes have changed, it skips the back end of compilation and linking, and proceeds directly to execution.



2.2.3. The Photoflo Runtime Manager

Photoflo management is partially the responsibility of the client, and partially the responsibility of the server. The client sends Execute, PutClientData, GetClientData, QueryPhotoflo, Await and Abort requests which the server must accommodate. Specific Procs in the XIE dispatcher receive these requests, and after possibly making other internal server calls (e.g., ProcExecute may call the compiler or linker first), route control to the Photoflo Manager, which resides in ddxie.



The entry points of each photoflo manager component are placed in the dixie data structures. Thus, the photoflo manager is purposely designed to support ample customization, at several different levels. For example, an md scheduler could be substituted, while to using other components from the mi photoflo manager. The selection of which components to use could be determined by the nature of the photoflo. For example, the general-case mi photoflo manager could be chosen if the DAG is topologically complex, whereas a simpler photoflo manager might be appropriate for linear photoflos.

2.2.3.1. Scheduler

The scheduler is responsible for managing the activation of elements in order to move data through the photoflo. The SI provides a default scheduler in the mi layer, which may be overridden by a device-dependent scheduler by the server implementor.

Any scheduler has the following responsibilities:

1. When data arrives from the client, it must be delivered to the appropriate receiving element.
2. Elements which have filled their output buffers must be suspended until there is room to generate more output.
3. Elements which have sufficient data to begin processing must eventually be activated, to avoid deadlock.

In addition there are the following issues to be dealt with:

1. Low-memory servers should minimize the requirement of large data buffers within the photoflo, possibly at the expense of some performance loss.
2. High-memory servers should maximize efficiency, possibly at the expense of using more memory in photoflo data structures.
3. For slow operations, it may be desirable to minimize the initial delay to receiving data in export elements.
4. For fast operations, it may be desirable to maximize performance, so as to increase the number of operations per second which can be carried out.

Consideration of the above issues and responsibilities leads to the notion of letting elements produce a configurable number of lines when activated. If an element only produces one line at a time and then returns to the scheduler for permission to proceed further, excessive overhead may result. If an element on the other hand waits for all input data to proceed and then produces all output data at once, excessive memory usage in the photoflo may occur.

The SI scheduler supports elements that can produce their output data in *strips*, that is, a fixed number of lines at a time. The number of lines in a strip will vary among elements and even servers, since memory/performance tradeoffs in the scheduler will tend to vary with the same tradeoffs in the server. Thus, the number of lines in a strip for any element is set by that element's ddx-level handler, at link time. At the same time, a minimum number of lines of input which are required to commence processing are also recorded in the element's *receptor* data structures.

Another concern of the scheduler implied by the above issues and responsibilities is to maintain an even flow of data through the photoflo. If the scheduler simply finds an element which can be activated and repeatedly executes it until all input data is exhausted, the data may wind up moving slowly through the photoflo in large chunks, as if the strip sizes were equal to the entire output image. Thus the SI scheduler attempts to balance the flow of data by reducing the priority of elements which have recently been activated, and increasing the priority of elements which have recently received data. The SI scheduler will also yield to the client when it has finished processing all elements since the last client yield, presuming there is more data which the client is obligated to send.

To summarize the heuristics upon which the SI scheduler are based:

1. elements produce output several lines at a time instead of one or all, to avoid thrashing in the first place and heavy memory consumption in the second.
2. the scheduler yields to Core X if there is pending client input data, after checking to make sure that all elements which are capable of processing have had a turn since the last yield.
3. elements which have recently been executed have their priorities downgraded.
4. elements which have recently received data have their priority upgraded.

Details of the actual algorithm employed for scheduling may be found in the last chapter on DDXIE Photoflo Management.

2.2.3.2. Data flo Manager

The data flo manager has the responsibility of moving data between the elements which produce it to the elements that consume it. In order to minimize overhead, pointers to the data are exchanged, rather than copying the data itself.

In this model, an element requests a pointer to an output strip from the data flo manager when it is ready to execute. The element maintains exclusive control over that strip while it is processing its data and placing it in the strip. Once the element has finished with the strip, it calls the data flo manager to pass it to downstream receiving elements.

After being informed that a strip has been completed, the data flo manager determines which downstream element receptors are in need of the data. A pointer to the strip is then added to the end of each such receptor's input strip list.

For the receiving elements, the strips will usually be read-only, unless the elements requests and receives permission to re-use the input strip for storing its output. Macros will be provided as a tool to hide the stripped nature of the data. While using these macros will avoid problems associated with downstream elements consuming strip output at different rates, elements will have the option of accessing data on a whole strip basis if they so desire.

Details of the actual algorithm employed for dataflo management may be found in the last chapter on DDXIE Photoflo Management.

2.2.4. The XIE Resource Manager

The XIE resource manager is responsible for creating, doing lookup, and destroying XIE resources. Those resources available to the client are Photomap, LUT, Photoflo, Photospace, ROI, and ColorList types, which are registered with the Core X resource manager to comply with the standard X model.

A photospace resource maintains a list of the photoflos which reside in it. When the photospace is destroyed, all of its photoflos are destroyed along with it. Destroying a photoflo requires going through its list of elements and destroying all of their data structures too.

2.2.5. The XIE Memory Manager

The XIE SI will layer an XIE-private set of allocation calls on top of the standard X allocation routines. This will permit customization of the memory manager to allow for special fragmentation avoidance algorithms, etc.

3. Extension Initialization

The image extension is initialized along with other extensions by setting the XIE flag in the module miinitext.c. This causes the function XieInit() to be called during server startup.

XieInit() calls CreateNewResourceClass() to register an XIE resource class, and registers the Photomap, LUT, Photoflo, Photospace, ROI, and ColorList resource types and their respective *deleteFunc* vectors by calling CreateNewResourceType().

Next XieInit() calls AddExtension() to register the XIE dispatchers and *closeDownProc* with Core X. AddExtension returns the major opcode, error and event bases for XIE to use.

XieInit() allocs a client array sufficient for holding MAXCLIENTS clients, and initializes the array to contain XieNullClient pointers.

The last step of extension initialization is to call XieReset(), which initializes XIE's internal state variables, if required.

4. Memory Management

The SI will place a veneer over the core X memory management routines (e.g. Xalloc, Xrealloc, Xfree, etc.). This will facilitate adding any special management hooks that may be necessary when the SI is ported to particular platforms. This will also facilitate monitoring memory usage, tracking down leaks, etc.

To cut down on memory management overhead during flo execution, the many data segment descriptors that might normally be allocated and freed as they are needed, will instead be returned to a look-aside list that is managed by the photoflo data manager (so that they can be more efficiently re-used). They will be returned to the underlying management routines (i.e. core X) when the flo completes (and if the number of cached descriptors exceeds a high water mark).

A global parameter will be used to hint at the most efficient alignment for scanlines, lookup tables, etc.. Individual elements are allowed to override this alignment hint in order to address their own particular alignment requirements (e.g. restrictions that might be imposed by an image processing accelerator).

5. Dispatching requests

XIE protocol requests are passed through the normal X protocol stream from the client to the core X *dispatch loop*. Core X dispatches XIE requests to the XIE dispatcher or the XIE swap-dispatcher. The XIE dispatcher of choice was selected at extension initialization time by AddExtension().

The XIE dispatcher passes the request directly to the appropriate *Proc* routine. The XIE swap-dispatcher must byte-swap multi-byte fields within the protocol request before calling the *Proc* routine. For requests that include PhotoElements, the element (and technique) fields must also be swapped. Conversion of IEEE-floats is deferred to the photoflo creation routines. Likewise, reformatting of client data (received with a PutClientData request) is deferred to the target element within the photoflo.

When the extension is done fielding the request, it returns a code value to indicate either success or failure. If the request is not a successful, an error packet is sent back to the client.

6. Data Representations

6.1. Canonic Data

To ensure the greatest processing efficiency the SI will maintain images using a limited number of convenient data types. Since data provided through PutClientData requests must be copied from the wire buffer before returning to core X, the least overhead is incurred if the data are converted as they are moved. All image data entering the SI from the client will be converted (if necessary) to one of the following data types:

- Bit: bi-level pixels. 8 pixels are packed into each consecutive byte
- Byte: pixels having 3 to 256 quantization levels, each pixel stored in the low-order bits of a byte (unused high-order bits within the byte are zero)
- Pair: pixels having 257 to 16,384 quantization levels, each pixel stored in the low-order bits of a byte-pair (unused high-order bits within the byte-pair are zero)
- Quad: pixels having more than 16,384 quantization levels, each pixel stored in the low-order bits of a 4-byte unit (unused high-order bits within the unit are zero)

An additional image data type arises from elements that can produce *unconstrained* data, which can only exist within the context of a photoflo. Unconstrained data must be converted, via a client specified Constrain element, prior to export out of the photoflo. The protocol makes no restriction on the representation of unconstrained data. In the SI, unconstrained data will be represented via " *typedef float Unconstrained;* " to facilitate redefining the representation.

Canonic form will take advantage of the natural byte order of the server architecture for data that spans multiple bytes. The SI will use Endian-intelligent macros to efficiently extract and insert arbitrarily aligned bit field data. There will be a module of conversion routines for converting to and from canonic form, and for promoting data to a *higher* canonic form (e.g. convert byte to float). Demoting data (e.g. from float to byte) is explicitly requested by the client via Constrain elements.

6.2. Non-canonic Data

All data that is transported through the protocol stream is considered to be *non-canonic*. Additionally, all encoded data is non-canonic.

6.2.1. Byte Stream Data

For data that are injected into a flo by a PutClientData request, conversions are specific to the target ImportClient element and decode technique. Since their parameters fully describe how the data are to be interpreted, the appropriate decode or conversion function is selected by the ddxie DAG analyzer. Similarly, it is the responsibility of an ExportClient element to format output data (as specified by client parameters) prior to it being sent to the client (the data are sent in response to a GetClientData request).

6.2.2. Compressed Data

Compressed data is stored as a byte stream, together with sufficient decoding parameters to decompress the data. These parameters include a decode technique.

6.2.3. Run-length Data

A ProcessDomain is not a resource, but rather the use of a photomap or ROI resource (or equivalent data imported directly from the client) in conjunction with an X,Y offset (V4.12 pg. 7-2). The Blend element is unique in that the process domain input data must be a *blend-plane*. A blend-plane is an n-level constrained alpha-plane (i.e. data that looks much like a gray scale image). All other elements that accept a process domain (e.g. Arithmetic, Compare, ..., Point, ExportClientHistogram) accept only ROI or *control-plane* type processing domain data. An ROI represents a *list-of-rectangles*, whereas a control-plane is treated as an array of Booleans.

Since some elements can accept both ROI and control-plane data types, their implementation can be simplified if these data types could be converted to a common form that offers processing efficiency. Using run-lengths, groups of image pixels could be alternately *processed* or *passed-through* without testing the domain on a per-pixel basis.

The SI will present control-plane and ROI data to an element's process domain input in run-length form on a per-scanline basis. Consecutive scanlines with identical sets of run-lengths can be counted as a vertical run. Encoding the data in this manner may also provide some compression, but that is not the primary consideration.

Conversion of *control-plane* data will be accomplished by inserting an additional element type between the source of the control-plane data and the process domain input of the element that requires it. This new element type will alternately count runs of zero and non-zero pixels in the image.

ROI data will be converted by the ImportClientROI element. The rectangles must be sorted by their Y offset and then by X offsets within the same Y, before they are converted into run-lengths.

Defining process domain data in this manner is akin to Y-X banding of regions within core X.

6.3. Tri-chromatic Data

Triple band data is stored and passed between flo elements in *band-by-plane* format (i.e. pixels for each band are segregated into separate image arrays). In XIE, the color space attributes of triple band data are known only to the client. Therefore, this data is not tagged as representing any particular color space.

6.4. Strips

One of the motivations for using photoflos is to reduce memory utilization by propagating segments of the image through each processing step to form the desired result, rather than consuming a full size image buffer at each step. If the segment size is too small, however, performance can suffer due to increased scheduling overhead. To address these conflicting problems, elements will produce a number of scanlines at a time, in *strips*, where the number of scanlines that constitutes a strip is a function of a tunable parameter.

7. Data Structures

7.1. Data Formats

A format-rec is used to describe the format of data that an element will produce or the format required by a receptor. It consists of:

- the data class (i.e. canonic, compressed, lookup-table, run-length)
- an in-line format-specific structure

Note: run-length data require no additional specification, byte stream data are described by client specified element parameters.

7.1.1. Canonic

A canonic-rec is used to describe the format of canonic image data that an element will produce or the format required by a receptor. For triple band data, a separate canonic-rec is used for each band. A canonic-rec contains:

- the canonic data type
- the band number
- the total width (in pixels)
- the total height (in pixels)
- the depth (in bits)
- the number of potential quantization levels
- the distance between adjacent pixels (in bits)
- the distance between adjacent scanlines (in bits)

7.1.2. Compressed

A compressed-rec is used to describe encoded image data that an element will produce or encoded data that is required by a receptor. A compressed-rec contains:

- the compression technique (either *decode* for data that is to be stored, or *encode* to describe how an export element should compress data)
- the band interleaving
- the band number
- the width (in pixels)
- the height (in pixels)
- the number of potential quantization levels
- a pointer to technique-specific parameters

  For triple band JPEG data, three compressed-recs are required to describe the individual bands despite the number of image arrays that are required to represent the data (1 or 3 depending on interleave).

7.1.3. Lookup-table

A table-rec is used to describe the format of lookup table data. It contains:

- the band order of lookup tables sent from the client
- the maximum number of quantization levels allowed in the input image (i.e. the number of entries per table)
- the number of quantization levels that may occur in the output image

7.2. Resource structures

7.2.1. ColorList

A colorlist-rec is used to define a ColorList resource. It contains:

- the resource-id of the colorlist
- the number of references to the colorlist (including 1 for core X)
- the resource-id of the colormap from which the cells have been allocated
- the number of cells allocated
- the pointer to the list of allocated cell indices

7.2.2. LUT

A lut-rec is used to define a LUT resource. It contains:

- the resource-id of the lut
- the number of references to the lut (including 1 for core X)
- the number of lookup tables (either 1 or 3)
- a format-rec containing the lookup table attributes
- pointers to the lookup tables

7.2.3. Photomap

A photomap-rec is used to define a Photomap resource. It contains:

- the resource-id of the photomap
- the number of references to the photomap (including 1 for core X)
- the number of chrominance bands in the image (1 or 3)
- per-band format-recs (i.e. queriable attributes: width, height, and levels)
- a pointer to the decode parameters
- a pointer to the image array for each band

7.2.4. ROI

A roi-rec is used to define a ROI resource. It contains:

- the resource-id of the roi
- the number of references to the roi (including 1 for core X)
- the length of the run-length encoded list of rectangles
- a pointer to the run-length list

7.2.5. Photospace

A photospace-rec is used to define a Photospace resource. It contains:

- the resource-id of the photospace
- the number of active photoflos in the photospace
- a queue head that points to the first and last photoflos in the photospace

7.2.6. Photoflo

A flo-def is used to define a photoflo resource. It contains:

- the Executable identity of the photoflo (i.e. the name-space/flo-id pair)
- Booleans: Active, Modified, ...
- the number of elements received from the client
- an array of pointers to client elements in phototag order
- a circular doubly linked list head that points to the *definition* DAG
- a circular doubly linked list head that points to the *optimized* DAG
- a flo-error-rec for passing error information back to dixie
- a flo-event-rec for sending event information back through dixie
- a pointer to ddxie-private data (the SI will hang a flo-tex structure here)
- a set of ddxie photoflo manager public entry points:
    - *link* — create a runnable photoflo, its elements, and data that persists between executions
    - *startup* — allocate/initialize data that are valid for a single execution instance and commence execution
    - *resume* — continue execution after the flo has yielded (not used in the SI)
    - *input* — push data from the client into a specified ImportClient element and resume execution
    - *output* — pull data for the client from a specified ExportClient element and resume execution
    - *query* — query a specified ImportClient element to see if it needs more data, or query a specified ExportClient element to see if it has data available
    - *abort* — stop execution and free single instance stuff
    - *destroy* — free persistent stuff and destroy the runnable DAG
- a set of ddxie photoflo manager private entry points:
    - *scheduler* — schedule elements to run until data exhausted or yielding is required
    - *data manager* — strip management

7.3. Photo Element Definition

A pe-def is used to define each client element supplied at photoflo creation. Each pe-def contains pointers for linking elements together according to their phototag inter-connections. A fully linked set of pe-defs constitutes a definition DAG that is suitable for presentation to ddxie for analysis. Each structure contains:

- the element phototag number received from the client
- the element type received from the client (within the optimized DAG this may identify additional converter or optimized element types that are defined for the server implementation)
- Booleans: loop-detect, orphan, modify, import, process, export
- a pair of pointers linking this element to the next and previous element in DAG order
- a pointer linking this element to another element of the same class (used to locate all the Import or Export elements; not used for Process elements)
- a pointer to link a client element to a replacement element in the optimized DAG
- a pointer to element specific parameters (e.g., parameters supplied by the client)
- a pointer to ddxie-private data (placed here by the ddxie *link* routine, removed by the ddxie *destroy* routine -- the SI will hang a pe-tex structure here)
- element entry points
  - create
  - initialize
  - activate
  - flush
  - reset
  - destroy
- fan out information:
  - the number of bands this element will produce
  - a format-rec describing each output band
    (dixie fills in attributes, ddxie fills in alignment ...)
  - the number of receptors connected to this element’s output
  - linkage to receptors of this element’s output
- the number of phototag inputs this element is using
  (i.e., the number of inputs from other elements, including process domain and lut inputs)
- an array of input connection information; each input connection includes:
  - the phototag of the source element
  - a pointer to the source element
  - the number of bands this input will receive
  - a format-rec describing each band
    (dixie fills in attributes, ddxie fills in alignment, ...)

7.4. DDXIE Structures

7.4.1. Photoflo run-time context

A flo-tex stores ddxie-specific information for the execution of a photoflo. It contains:

- a pointer back to the flo-def
- information for the scheduler (e.g. the ready-list)
- information for the data manager (e.g. a look-aside list of strips that can be re-used)
- a pointer reserved for yielding information

7.4.2. Photo Element run-time context

An pe-tex stores ddxie-specific information for an element. It contains:

- a pointer back to the pe-def
- a pointer back to the flo-tex
- output strip size
- a pointer to an array of receptor-recs *
- a pointer to additional storage for element-private information *

  * the structures pointed at by these pointers are contiguous with (appended to) the pe-tex.

7.4.3. Receptor

A receptor-rec provides an execution context for data that are to be brought into an element. It contains:

- a pointer to the pe-def that is supplying the data
- the minimum number of data units needed for activation (per band)
- the total number of data units currently available (per-band)
- a circular doubly linked list head that points to the strips that are available (per band)

7.4.4. Strip

A strip describes a segment of data that is produced by an element and is then given to the data manager for distribution to one or more receptors. A strip describes an array containing one or more complete data units (for canonic data a unit is a scanline). It contains:

- forward and backward pointers to the next and previous strips
- a pointer to the format-rec that describes the data format
- a reference count
- the number of data units in the strip (e.g. number of scanlines for canonic data)
- the first scanline number (if canonic)
- the total size of the data array (in bits)
- the offset within the data array to the first data (in bits)
- the base address of the data array

8. Protocol requests

Each of the XIE protocol requests is listed below. For the sake of brevity, information that can be found in the protocol document (V4.12) is not duplicated here. Instead there is:

- a short description of each request
- an overview of the actions taken for a successful request

If a request cannot be honored due to an error, an error packet will be returned to the client, as prescribed in the protocol document.

Each protocol request will have a dedicated *Proc* and *SProc* routine which is called from the XIE dispatcher (which is called by the X dispatcher). Swap routines will also be provided for replies, errors, and events.

8.1. Global Query requests

8.1.1. QueryImageExtension

Each client is required to invoke QueryImageExtension prior to issuing any other XIE protocol requests in order to establish protocol version compatibility between client and server.

The xie-client-table is checked for an existing xie-client-rec for the current client. If the client is not found (as is the case when QueryImageExtension is used for XIE connection setup) a new xie-client-rec structure is added to the table.

Applying the compatibility rules spelled out in the protocol document, the XIE version is chosen and saved in the xie-client-rec. The set of *Proc* and *SProc* routines that match this version are also saved in the xie-client-rec.

QueryImageExtension generates a protocol reply which returns the chosen XIE version information, the ServiceClass, and the Alignment capabilities of this server.

8.1.2. QueryTechniques

QueryTechniques provides information about a selected group of techniques that are supported by the server. This is the only method by which a client can discover implementation specific technique information such as:

- what are the default techniques?
- which techniques require parameters?
- what is the relative execution speed?
- what private techniques have been added?

A reply is returned to the client containing a TechniqueRec for each technique in the specified group.

8.2. Resource requests

8.2.1. Overview

The ColorList, LUT, Photomap, and ROI resources provide server-side storage of XIE data types. Each of these resource types is registered with core X at extension initialization time (CreateNewResourceType) along with a resource specific *deleteFunc* (e.g. DeletePhotomap). This function enables core X to destroy these resources if the client connection is closed.

Chapter 4 of the protocol document explains resource creation, destruction, and flo bindings using a *bucket* and *handle* analogy. The following sections explain how these requirements will be satisfy.

8.2.1.1. Creating Resources

Each of these resources is created via an explicit XIE protocol request. Upon creation each resource has a reference count of one, but has no resource specific data or attributes. If the resource is created successfully, AddResource() is called to register it with core X.

These resources are populated with data and attributes as the result of executing a flo. With the exception of the ColorList resource, these resources can be used to provide input for a flo.

8.2.1.2. Flo'ing Resources

When a flo becomes *active*, the reference count of each XIE resource referenced by the flo is incremented once for each reference to the resource.

For LUT, ROI, and Photomap resources, it is permissible for multiple elements in an *active* flo (or different *active* flos) to access the same resource simultaneously (e.g. the explanation of pseudo in-place operations). Multiple simultaneous writes to the same ColorList are illegal. Therefore a ColorList's reference count can never exceed two.

If multiple elements within a flo export to the same resource, upon successful flo completion the resource's *handle* will be connected to one of the new *buckets* (the protocol does not specify which bucket will win). If multiple flos are exporting to the same resource, each flo will leave a new *bucket* attached to the resource upon successful completion. Synchronizing the execution of multiple flos is an exercise left to the client.

When the flo leaves the *active* state (because of done, error, or abort), the *deleteFunc* of each referenced XIE resource is called to decrement the reference count (once for each reference). If a resource has zero references, its structure (and everything associated with it) is freed.

8.2.1.3. Destroying Resources

Each of these resources can be destroyed via an explicit XIE protocol request which indirectly calls the resource's *deleteFunc* via the core X FreeResource() routine. Calling the *deleteFunc* in this way disassociates the resource-id from core X. The *deleteFunc* is specified at extension initialization time via CreateNewResourceType().

Other responsibilities of the *deleteFunc* are to zero the resource-id in the resource structure and decrement the reference count. If the count goes to zero, the resource structure and everything associated with it are freed, otherwise this action is delayed until there are no outstanding references.

8.2.2. ColorList

8.2.2.1. Create

Create an empty color-list-rec and call AddResource(). A ColorList is a resource used by the ConvertToIndex element to record colormap cells that it allocates. These cells can be freed by explicit ColorList requests or via other mechanisms that are specified in the protocol document.

8.2.2.2. Query

Reply with the list of colormap cells currently referenced by the ColorList. If there are none or the colormap has been freed, the colormap and count fields of the reply will both contain zero. If the query is made while a flo is actively allocating cells, the query will return a snapshot of the ColorList at the time of the query.

8.2.2.3. Purge

If the reference count is one, all cells that are currently allocated are freed, thus restoring the color-list-rec to its create-time state.

8.2.2.4. Destroy

Call FreeResource() to disassociate the ColorList resource-id from core X. The registered *deleteFunc* may purge cells held by the ColorList and free the color-list-rec (see <u>Destroying Resources</u> above).

8.2.3. LUT

8.2.3.1. Create

Create an empty lut-rec and call AddResource(). A LUT is a resource that contains lookup-table array(s) that are used by the Point element. A LUT resource is populated via an ExportLUT element. Its attributes are inherited indirectly from an ImportClientLUT element (XIE does not provide any facilities for synthesizing exportable LUT data in the server).

8.2.3.2. Destroy

Call FreeResource() to disassociate the LUT resource-id from core X. The registered *deleteFunc* may free the lookup arrays and the lut-rec (see <u>Destroying Resources</u> above).

8.2.4. Photomap

8.2.4.1. Create

Create a photomap-rec and call AddResource(). A photomap is a resource that is used to store a complete image. Data stored in a photomap may be imported into a flo and used as image data for *display*, or exported from the flo, or used as a process domain.

The photomap-rec is populated by an ExportPhotomap element. This export element will receive the image from a process element, or an import element (i.e. from the client, a photomap, or a drawable). Image attributes are also inherited from this element.

8.2.4.2. Query

Generate a reply that reflects the current set of image attributes. If the photomap has not been populated (i.e. the number of bands is zero), all fields of the reply will contain zeros and *populated* will be false.

8.2.4.3. Destroy

Call FreeResource() to disassociate the photomap resource-id from core X. The registered *deleteFunc* may free the image descriptors, image data, decode parameters, and photomap-rec (see <u>Destroying Resources</u> above).

8.2.5. ROI

8.2.5.1. Create

Create a roi-rec and call AddResource(). An ROI is a resource which contains a list of rectangles that are used to define a process domain. An ROI resource is populated via an ExportROI element.

The list of rectangles received from the client is sorted and converted into run-lengths by the ImportClientROI element (this is explained in the section on <u>Processing Domains</u>). This data is stored in the roi-rec by an ExportROI element. ROI data always originates from the client, XIE does not provide any facility for synthesizing exportable ROI data within the server.

<u>Note</u>: the list of rectangles returned from ExportClientROI will be a list of sorted non-overlapping rectangles (reconstructed from the run-length list). Although the contents of this list may differ from the list received from the client, it will be logically equivalent. The protocol does not specify that the original list must be returned.

8.2.5.2. Destroy

Call FreeResource() to disassociate the ROI resource-id from core X. The registered *deleteFunc* may free the run-length list and the roi-rec (see <u>Destroying Resources</u> above).

8.3. Photoflo related requests

Photoflos provide the mechanism by which a sequence of import, process, and exports steps are described. Two photoflo modes are supported: an immediate flo is executed once and then it is destroyed, whereas a stored flo can be executed repeatedly and offers the option of modifying its parameters between executions.

8.3.1. ServerIDspace

Since stored flos can be re-executed, they generally enjoy a *normal* life expectancy. Therefore they are identified using the same name-space as other resources created by the client. Their resource-ids can be re-used however, by *redefining* the complete contents of the flo.

8.3.2. Photospace

Immediate flos may be generated in rapid succession (and are short lived relative to *normal* resources). Therefore, a separate 32-bit wide identifier name space is provided for their execution.

8.3.2.1. Create

Create a photospace-rec and call AddResource(). Since only a small number of immediate flos are likely to be active in a photospace at any given time, the SI will maintain the active set of flos as a doubly linked list.

8.3.2.2. Destroy

Call FreeResource() to disassociate the photospace resource-id from core X. The registered *deleteFunc* will abort and destroy all immediate flos known to the photospace and then free the photospace-rec.

8.3.3. Immediate-mode specific

An immediate flo begins execution as soon as the protocol request containing the sequence of steps to be performed is received. Upon completion the flo is automatically destroyed.

8.3.3.1. ExecuteImmediate

Create a flo-def and copy the supplied list of elements to the flo-def while verifying that the list is a DAG. Next, the flo-def is added to the specified photospace and execution commences. Upon completion the flo is destroyed and the flo-def is removed from the photospace. Flo creation, execution, and destruction is covered in detail in the chapters on photoflo management.

8.3.4. Stored-mode specific

8.3.4.1. Create

Create a flo-def, copy the supplied list of elements into pe-defs (while verifying that the list is a DAG), and then call AddResource(). This process is covered in detail in the chapter on <u>DIXIE Photoflo Management</u>.

8.3.4.2. Modify

After verifying that no topological or element type changes are being attempted, the parameters of the specified set of elements are replaced. These elements are flagged to note that they have been *modified*. The chapters on photoflo management address the additional side affects that may be incurred when a previously executed flo is modified.

8.3.4.3. Redefine

The SI will simply call FreeResource() to destroy the old flo in its entirety, and then create a new one using the old resource-id.

8.3.4.4. Execute

Prior to executing a photoflo several preparatory steps are necessary:

- each element's parameters and data attributes are validated
- the DAG is analyzed to choose the most appropriate manner in which to execute it
- an optimized DAG is generated, if any refinements to the definition DAG can be made
- the required sequence of elements is instantiated

Finally, the flo is set *active* and execution begins. Flo execution is covered in detail in the chapters on photoflo management.

8.3.4.5. Destroy

Call FreeResource() to disassociate the flo's resource-id from core X. The registered *deleteFunc* will abort execution (if active), free everything attached to the flo, and then free the flo-def.

8.3.5. ClientData transport

8.3.5.1. Put

PutClientData supplies a byte stream of data from the client that is to be delivered to a specific ImportClient element within a flo. Since the interpretation of the data is specific to the target element and its parameters (and possibly a technique), any reformatting of the data to convert it to canonic form (including byte swapping) must be controlled by the receiving element.

8.3.5.2. Get

GetClientData generates a reply which returns a byte stream of data from a specific ExportClient element within a flo. The data will be formatted according to the parameters specified to the supplying element prior to the commencement of flo execution.

8.3.6. Query/Control

8.3.6.1. Query

Reply with the status of the specified flo (*active*, *inactive*, or *nonexistent*). If it is *active*, two lists of elements are provided: one listing elements waiting for data from the protocol stream, and the other listing elements which have data for the protocol stream.

8.3.6.2. Await

If the specified flo is not found in the *active* state, no operation is performed. Otherwise IgnoreClient() is called to block further requests from the client. Later, when the flo transitions out of the active state, AttendClient() will be called to allow blocked requests to be processed again.

8.3.6.3. Abort

If the specified flo is not found in the *active* state, no operation is performed. Otherwise execution of the specified flo is terminated and output pending for the client is destroyed. A stored flo is returned to the *inactive* state; an immediate flo is destroyed.

9. DIXIE Photoflo Management

9.1. Overview

The protocol representation of a photoflo is a list of elements. There are three element types: import elements which bring data into the flo, processing elements which operate on the data, and export elements which transfer data out of the flo to external resources or back to the client.

Connections between elements is accomplished through the use of phototags. An element's relative position on the protocol list determines the element's phototag number. The source(s) for each element are explicitly specified using the phototags of upstream elements. The output of an element can fan out to any number of downstream elements.

Dixie is responsible for creating the photoflo data structures that will contain the client's list of elements. The same structures are used by *stored* and *immediate* flos. At create time, dixie also converts the client list into a fully connected DAG. Dixie relies on ddxie to analyze the DAG in order to select an appropriate method to execute the sequence of operations. Ddxie fills in a vector of entry points within the flo-def, so that dixie can control the overall photoflo execution process. A call through the *link* entry point generates a re-usable execution context (called a flo-tex) and an associated set of run-time element handlers. The *startup* entry point (re)initializes the flo and starts execution. *Input* and *output* entry points are used to move client data in and out of the flo on behalf of data transport protocol requests. Additional entry points are provided for restarting the flo after it has yielded, for obtaining status information, for stopping the flo prematurely, and for destroying it.

The following sections will discuss photoflo management from the dixie point of view. The chapter titled <u>DDXIE Photoflo Management</u>, describes the actions performed within ddxie.

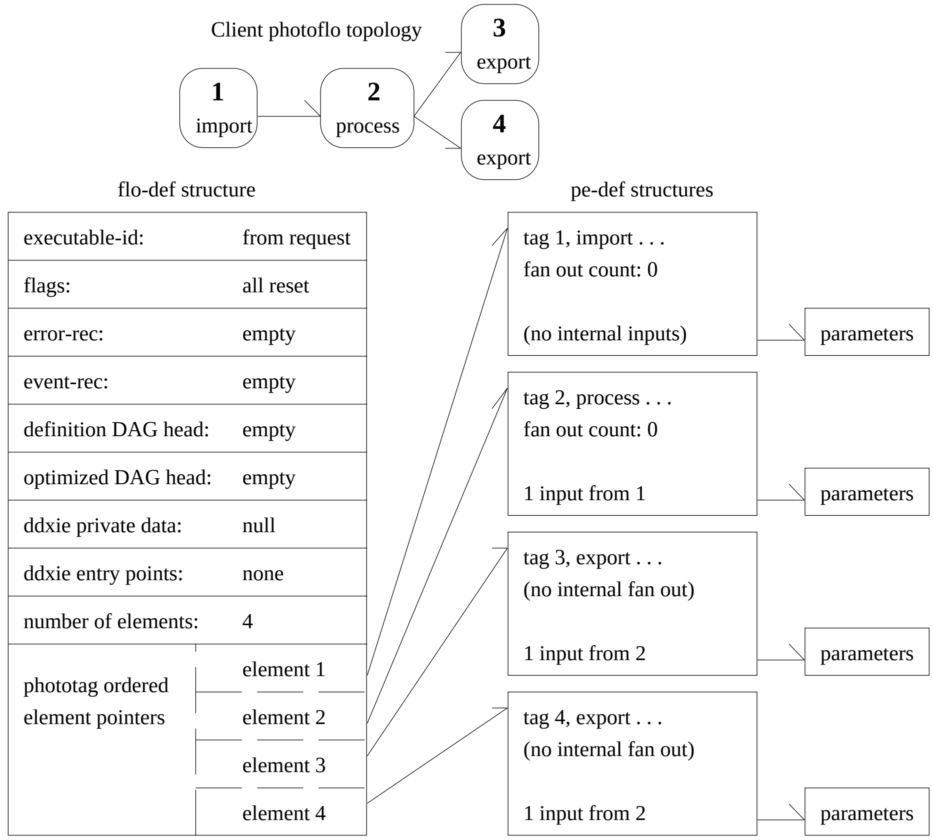
9.2. Photoflo Creation

9.2.1. Creating the Flo

When the server receives an ExecuteImmediate or CreatePhotoflo protocol request, a flo-def is allocated and initialized. The list of elements received from the client is parsed in the order received. The following actions are performed (within dixie) for each element:

- verify that each phototag input is specified within the bounds of the client element list
- create an pe-def large enough to accommodate the number of phototag inputs
- store element information in the pe-def
  (e.g. the element's phototag index, class, type, and input phototag connections)
- store the element's parameters (including technique data) in a separately allocated area
  (any necessary floating point conversions are performed at this time)

The address of each pe-def is stored in phototag order within the flo-def. Upon completion of this initial creation phase the photoflo looks like:



Initial Photoflo creation showing linkage to phototag ordered elements and client parameters

9.2.2. Traversing the DAG

The next phase of photoflo creation converts the list of client elements into a DAG. The element list is scanned in phototag order. At each export element, the phototag input connections are followed *upstream* via a recursive depth first search algorithm until one of the following conditions occurs:

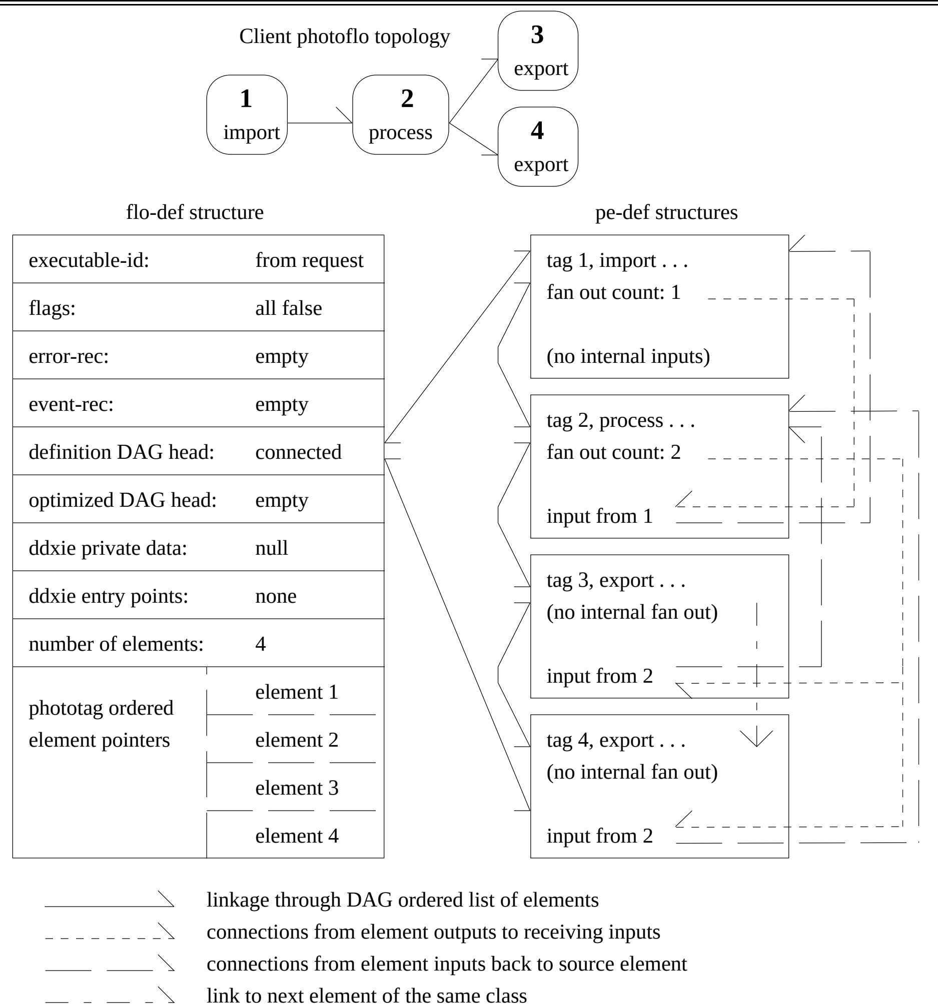
- an import element is found
- an element with fully connected inputs is encountered
(i.e. one that was examined while searching back from a previous export element)
- a topological error condition is detected

Topological errors are discovered during the DAG definition process by setting a flag in the current element prior to invoking the recursive search routine to examined the element's inputs, and then resetting the flag when all the input elements have been examined. A loop condition exists if a flagged element is found during the recursive search. Attempting to use an export element as a data source is also a topological error.

As the search algorithm considers each element, the following actions are performed:

- if the element being examined is not an import element:
    - set the *loop-detect* flag in the current element
    - invoke the search algorithm to examine each of the element's inputs
      the inputs of multi-input elements are searched in the following order:
        1. lut input
        2. process domain input
        3. source input(s) 1 through n
    - clear the *loop-detect* flag in the current element
- link the current element's output and the *downstream* element's input to each other
- append the current element to the definition DAG

Continuing with the example given above, the definition DAG structures would be linked like this:



Completed Photoflo creation showing definition DAG linkages

The definition DAG is a circular doubly-linked list of pe-defs that has the following characteristics as seen from the DAG head:

- the first element in the forward direction is an import element
- the first element in the backward direction is an export element
- for each element that is examined in a forward traversal, all of the elements from which it obtains input have already been examined

Depending on which export element was considered first, the DAG order of our simple flo illustrated above would be either:

1, 2, 3, 4, or 1, 2, 4, 3.

If the export elements in the flo illustrated below were considered in ascending order (i.e. search back from element 2, then from 5, etc.), the DAG ordering would be:

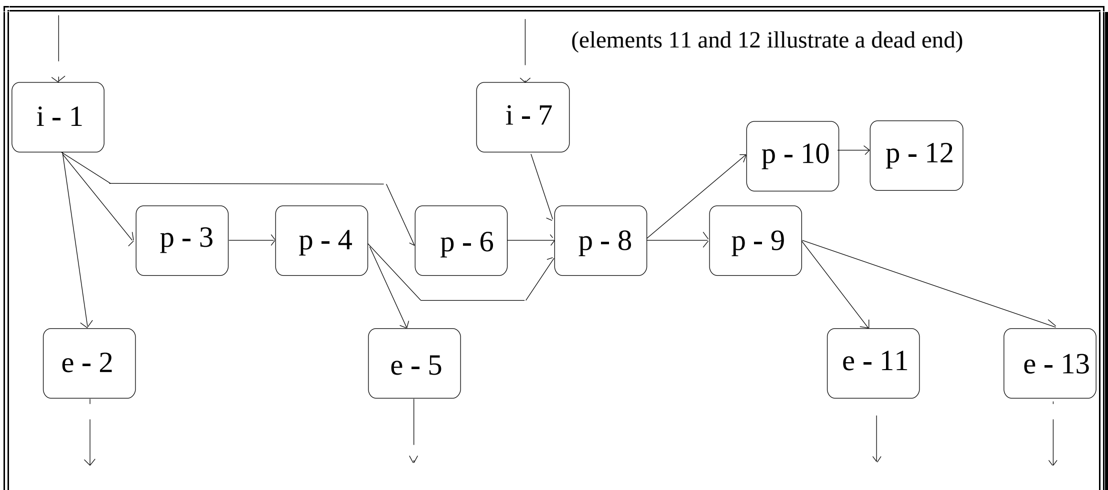
1, 2, 3, 4, 5, 7, 6, 8, 9, 11, 13.

If, however, the export elements in the flo below were considered in descending order (i.e. search back from element 13, then from 11, etc.), the DAG ordering would be:

7, 1, 6, 3, 4, 8, 9, 13, 11, 5, 2.

The DAG characteristics described above hold true for any sequence in which the export elements could be considered.

It is possible that the output from some elements will not terminate at an export element. The protocol states that these elements should be ignored. Therefore, elements 10 and 12, in the figure below, would not be included in a definition DAG because they do not terminate at an export element.



Modified example photoflo taken from figure 5-2, X Image Extension Reference Manual, v4.12

9.2.3. Registering the Flo

Provided no errors are encountered while creating and validating the photoflo, the flo-def structure of an immediate flo is registered with the specified photospace, whereas AddResource() is called to register a stored flo with core X.

9.3. Photoflo Execution

9.3.1. Prepping the DAG

Before a flo can be made active, some preparatory steps are necessary. First the DAG is traversed in the forward direction to execute each element's dixie prep routine which will:

- lookup external resources (colormap, colorlist, drawable, gc, lut, photomap, roi, ...)
  (for XIE resources their reference counts are incremented)
- validate the attributes of each source of data for the element
- validate the element's parameter values (including technique data)
- determine what the element's output attributes will be
- determine if the flo has been *modified* since a previous execution
  - the element was affected by a ModifyPhotoflo request
  - source attribute changes have caused output attributes to change

If there is an executable photoflo that already exists and it has been modified in such a way that re-compilation is necessary, the photoflo manager's destroy function is called, in preparation for a new compilation.

9.3.2. Analyzing the DAG

When the elements have been prepped by dixie, the DAG is suitable for presentation to the DAG analyzer. The purpose of the analyzer is to define an appropriate solution to the problem to be solved (e.g. coalesce multiple elements into a single *super-element*, insert data type conversion elements, insert data alignment elements, ...), and to choose a photoflo manager that can execute it.

If a previously executed flo still exists, the analyzer is not called. The current executable will be re-used because no additional analysis is required.

If an optimized DAG has been generated, the analyzer will attach it to the flo-def. The analyzer establishes a photoflo manager by placing its *link* entry point in the flo-def.

9.3.3. Linking an executablePhotoflo

The first time a photoflo is executed, the flo-def *link* function is called to generate the run-time structures for the photoflo. It must:

- create a run-time context for the flo manager (attach a flo-tex to the flo-def)
- create a run-time context for each element (attach a pe-tex to each pe-def)
- establish the remaining entry points in the flo-def

9.3.4. Starting the flo

The flo-def *startup* function is called to initialize the flo and begin execution. The photoflo manager startup routine will initialize each of its elements (in DAG order), and call upon the scheduler to run the flo. The next chapter covering ddxie photoflo management describes the actual execution of the flo.

If there are no ImportClient or ExportClient elements in the flo, it can run to completion. Otherwise, execution is dependent on data transport protocol requests.

9.3.5. Resuming the flo

When a PutClientData request is received, the flo-def *input* function is called with the phototag of the target element and a pointer to the available data. Calling *input* gives the new data into the specified element and causes the scheduler to resume flo execution.

When a GetClientData request is received, the flo-def *output* function is called to fetch some data from the specified element. This may cause the scheduler to resume execution (e.g. execution may have been suspended due to the output buffer being full).

A *resume* entry point is provided in the flo-def for implementations that support *yielding*. There will be no *resume* routine supplied with the SI.

9.3.6. Stopping the flo

Dixie does not get involved in determining when a flo reaches completion. Rather ddxie is responsible for determining if all data has been processed and if any data transport requests are required for completion.

If an error is encountered within the ddxie flo manager, the error detail is recorded in the error- rec, which is embedded in the flo-def, and control is returned to dixie. When an error is discovered by dixie, the flo-def *abort* function is called to stop the flo. Finally, the error is sent to the client via core X mechanisms.

If an Abort request is received from the client, the flo-def *abort* function is called to stop the flo.

9.3.7. Destroying an executable Photoflo

Dixie destroys a flo-tex by calling the flo-def *destroy* function which allows ddxie to free its resources and remove the ddxie-private flo-tex pointer from the flo-def.

9.3.8. Event notification from a flo

Several elements can generate events, as can flo completion. The flo-def includes an event-rec where details about the event can be recorded. Ddxie calls the dixie SendEvent() routine to request that the client be notified.

10. DDXIE Photoflo Management

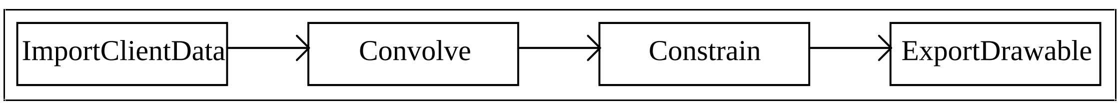
Ddxie contains a DAG *analyzer* that defines an appropriate solution for the image equation expressed by the DAG, a *linker* that creates an executable sequence of element handlers to solve the problem, and a *flo manager* to produce the result. The element handlers, which also reside in ddxie, are discussed in the next chapter.

10.1. DAG Analysis

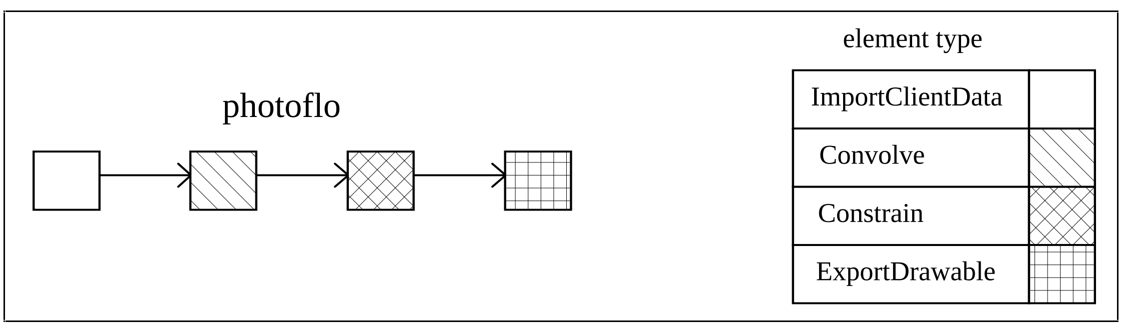
The purpose of the DAG analyzer is to define an appropriate solution to the image equation expressed by the DAG (e.g. choosing an element handler for a specific technique, coalescing multiple elements into a single "super-element", or inserting data type conversion elements), and to choose a photoflo manager that can execute it.

10.1.1. Choosing Element Handlers

The first step is to select an appropriate element handler for each element. To better understand how this is performed, an example is shown below. Assume that each photoelement type defined in the XIE protocol is assigned a different shading pattern. Then the flo



could be represented as:



For each element type (shading pattern), the mi layer of the SI will provide a set of handlers which can implement the computation specified by that element. A single handler may suffice for elements which do not take techniques, whereas an element like ImportClientData, which can take a number of different techniques, might be implemented as several handlers. Besides having handlers to account for technique variations, it may also be convenient to separate handlers on the basis of import type restrictions, machine dependence, and degree of optimization.

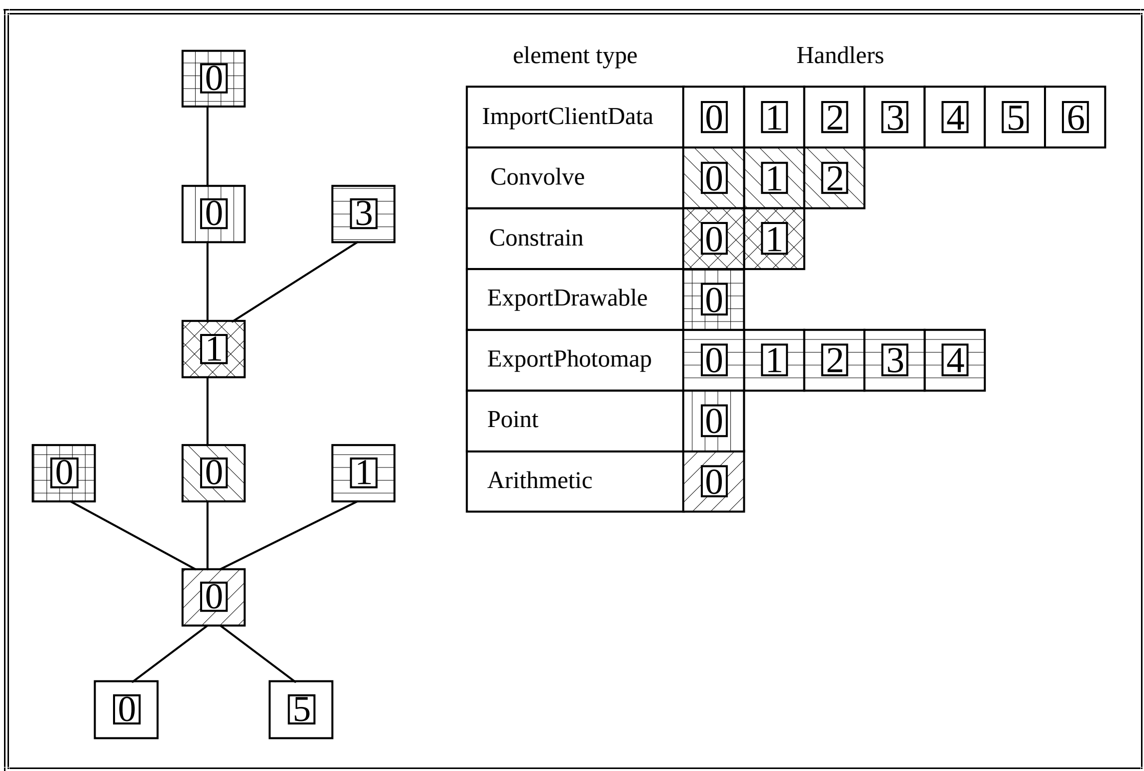
Given the framework established above, code generation for a photoflo may be viewed as a labeling problem, in which it is necessary that the DAG analyzer find a valid label for each element in the flo. The analyzer is only allowed to label an element with a handler of like shading pattern.

All handlers will output a type which is the smallest acceptable canonical type for the input data and the particular operation. Import elements will thus be able to write down their output image data types immediately.

Let k be the distance from an element to an Import element. Given a flo which has been labeled and had types resolved up to a distance of k-1, the DAG analyzer will label and resolve all elements of distance k by the following method:

For each element E_{ki} of distance k:

- For each element $E_{ki,j}$ which feeds that element:
  - Propagate the known output type of $E_{ki,j}$ to the corresponding input receptor on E_{ki} (these will be treated as "suggested input types" by the handlers).
- Call HandlerTest, giving it the flo-def and pe-def. A handler will be selected which resolves all input types. If any input types match, the selected handler will insert a labeled ConvertType pseudo-element into the flo.
- Label E_{ki} with the selected handler.



The HandlerTest procedure simply goes through the list of handlers which have the same shading pattern as the element in question, and calls the handler's *test* entry point. The test routine checks to see if this handler can accommodate the parameters of the element (obtained from the pe-def) in combination with the input data types. If the handler rejects the element, the HandlerTest procedure checks the next handler of the same shading pattern in the list. The SI element handler sets will be written so as to ensure that regardless of what the input data type is, at least one handler per shading pattern will accept any valid element.

Handlers have the option of accepting seemingly non-canonical data types by inserting a ConvertType pseudo-element in the path between the input element and the receiving element. This ConvertType element must be able to convert any non-canonical data type to a canonical type. The handler doing the insertion specifies the type to which it wants the data to be converted. This mechanism allows the mi handlers to continue working even if a server implementor chooses to invent new canonical types. Thus, the criterion that there must be at least one valid handler for any element will remain fulfilled even if new types are introduced.

10.1.2. Choosing a Linker andPhotoflo Manager

The analyzer must establish a linker and photoflo manager by placing their entry points in the flo-def. The mi linker and photoflo manager can handle all element handlers (whether mi or md) that abide by their interface conventions.

If an md analyzer is substituted for the mi analyzer, it may elect to use a *foreign* linker, photoflo manager, and element set (e.g. the pipe manager and sample elements contributed by HP). In this case, the foreign linker would need to translate the contents of the pe-def structures into its own set of elements.

10.2. Photoflo Linker

The linker is invoked by dixie calling through the flo-def link entry point (it is not called when an unmodified photoflo is re-executed). The linker creates a run-time context (a flo-tex) for use by the flo manager’s components. It is also responsible for the creation and static initialization of the element handlers selected by the DAG analyzer. This is accomplished by calling the create entry point (stored in each pe-def) of each element selected.

10.3. Photoflo manager

The photoflo manager is a collection of component parts that are called upon from dixie to perform specific tasks during the startup, execution, and eventual tear down of a photoflo.

10.3.1. Startup

When dixie calls through the flo-def startup entry point, the flo manager calls upon its sub-components to (re)initialize their run-time context. The flo manager also calls through the initialize entry point of each element to allow the elements to (re)initialize their run-time context. Finally, it transfers control to the scheduler to run the flo.

10.3.2. Scheduling

The scheduler is responsible for running available data strips through elements. The scheduler is initially called as the final step of photoflo startup. Later, as data arrives from the client, or data is requested by the client, the scheduler is started to satisfy the request.

Scheduling of individual elements for execution is handled through a *ready-list*. The first time a photoflo is executed, only ImportResource elements have data. The ready-list is initialized with these elements. Starting from the front of the ready-list, each of the elements on the ready-list are executed in order. An element will produce a maximum of one strip of data each time it executes*. This strip is passed to the data manager and then on to appropriate downstream receptors. Elements receiving data from this element are examined to see if they have sufficient data to execute. If they do, they are pushed onto the front of the ready-list. If the current element was an import element it is removed from the ready-list, otherwise it is allowed to remain on the ready-list (assuming it has more data to process) so that it may resume execution after the downstream elements have consumed its output.

One key concept to this algorithm is the removal of elements from the ready-list. Once an import element executes and produces up to one strip of data, it is removed from the ready-list, even if more data is available. Other element types are not removed from the ready-list until they can proceed no farther. This method of removing elements from the ready-list, combined with prepending downstream elements to the head of the list, has the effect of pushing strips of data through the flo before new strips are brought in from import elements.

When all data has been processed, the import elements are re-examined. If there are any ImportClient elements, control is returned to Core X. This is done to prevent large amounts of server data (possibly whole images) from being buffered in the flo while the client is stalled. If only ImportResource elements are found, they are put back on the ready-list and execution continues, as described above, until the flo has completed.

Each subsequent execution of the flo will be in response to the client. The element indicated in the protocol request, and all ImportResource elements, are placed on the ready-list. As before, each time an import element is executed, it is removed from the list and all downstream elements connected to its output with

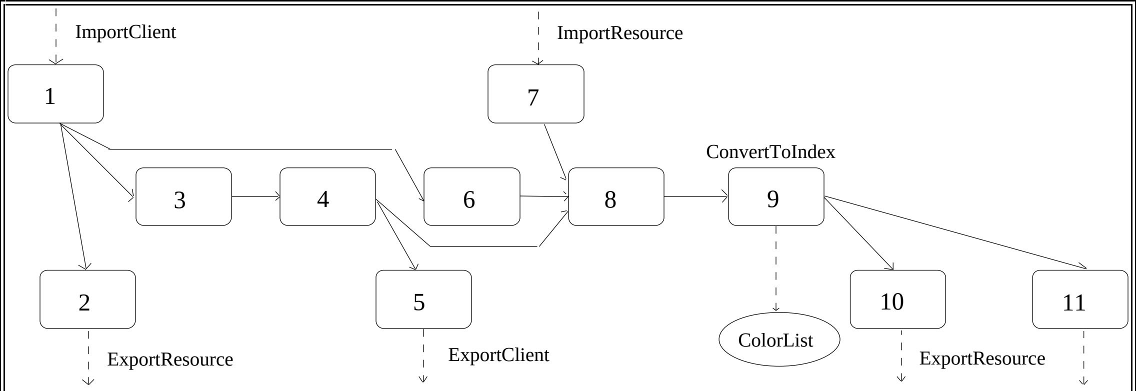
---

* There are three exceptions:

- The entire buffer of data received from the protocol stream for an ImportClient element will be presented as a strip of non-canonic data. The element is then responsible for converting this byte stream of data into canonic form, and processing or buffering the data prior to allowing control to be returned to core X.
- Export elements pass data out of the flo allowing them to process all data available when executed.
- LUT arrays are indivisible units. Therefore, a strip containing a lookup table will contain the entire array for a single band.

sufficient data to execute are pushed onto the front of the ready-list. If an import element was the target of a PutClientData request, it is re-examined to see if it has more data available. If so, the ready-list is re-initialized and execution continues (as above). If no more client data is available, control is returned to Core X.

As an example of how the scheduler would process a flo, the sample flo from Figure 5-2 of the protocol document (V4.12) is shown below.



Example photoflo taken from figure 5-2 in version 4.12 of
X Image Extension Protocol Reference Manual

This flo has two import elements, 1 and 7. Element 1, is an ImportClientPhoto element. Element 7, importing a Photomap, is an ImportResource element. When the flo is first executed, only element 7 will have data. Element 7 will import one strip of the photomap and cause element 8 to be placed on the ready list. Since element 7 is an import element, it is removed from the list.

Element 8 can now process the data and pass it on to element 9's receptor. Element 9 is a ConvertToIndex element that is using a technique which requires the entire image before it can continue. Since element 9 does not have sufficient data to execute, it is not added to the ready-list.

Assuming that element 8 has processed all of its data, it is removed from the list. With an empty list, the scheduler examines the two import elements. Noting that at least one ImportClient element is present, control is returned to Core X.

After the initial execution cycle, the flo will only be executed when data is received from the client for an ImportClient element. When such data arrives, the target ImportClient element (element 1) and all ImportResource elements (element 7) will be placed on the ready-list.

After element 1 has generated a maximum of one strip of data, the following elements would be on the ready list (elements 2, 3, and 6 all have sufficient data to execute with element 1's output):

2 3 6 7

Element 2 would be executed next. Since it is an export element, it will exhaust all data passed to it and be removed from the list. Element 3 will be processed next. Following its execution, element 4 will be added to the front of the list and executed, causing elements 5 and 8 to be added to the list. Assuming that elements 3 and 4 both have no further data to process, the resulting ready-list is:

5 8 6 7

Element 5 is an export element and passes all data it receives. It is then removed from the list. Element 8 processes data available to it from elements 4 and 7. Since element 9 is still unable to execute, it is not added to the list. This leaves the following ready-list:

6 7

With the execution of element 6, element 8 is again added to the ready list. After element 6's data passes to element 9's receptor the list is left with only element 7. Element 7 is executed to load another strip of data from the server photomap. At this point, control is returned to Core X. The photoflo waits until it is reactivated by more data being received from the client for element 1.

This process repeats until all data has been received from the client. Element 9 can then execute, which in turn allows export elements 10 and 11 to execute. Element 9 stays on the list until it exhausts its data. At this point, the photoflo operation is complete.

10.3.3. Data Manager

It is the responsibility of the data manager to coordinate the flo of data between element outputs and inputs. Since data is processed in strips, it will often be the case that data required by a downstream element will span two or more buffers generated upstream. To eliminate the complexity and overhead of concatenating data segments, data accesses within elements can be conducted on a line by line basis, while data transfers between elements are conducted on a strip basis.

When an element is producing a strip of data, the buffer it is writing its data into is private to the element. Only when the strip is passed back to the data manager is it made visible to downstream elements for processing. When the completed strip of data is passed to the data manager, it is linked to a list of strips maintained by each downstream receptor. As an element requires data, it "reads in" the data by transferring strips from its receptor into its private data section. The element may then access individual lines of data within the strip through a set of light-weight macros. These macros are provided as a tool to simplify the access of scan lines within strips but do not preclude elements from operating on data in strip form. When the element is done with the strip, it informs the data manager.

The strip itself contains a count of the number of references to the strip. When a strip is assigned as an element output buffer, this count is 1. When the completed strip is returned to the data manager the reference count is set to the number of receptors that the strip is transferred to. When an element asks the data manager for an available strip, the reference count is unaffected until the element notifies the data manager that it is done with the strip.

When an element finishes reading from a strip (or it has an output strip that it does not wish to propagate), it informs the data manager and the reference count is decremented. If the reference count is decremented to 0, the strip is either freed or cached for re-use.

The data manager also provides a mechanism whereby in-place operations may be performed within the strip. When this is allowed, only a single element has access to the strip. Since it is using it for both read and write, the reference count is 2. When it completes the strip it must give it to the data manager for transfer downstream and notify the data manager that it is done reading it.

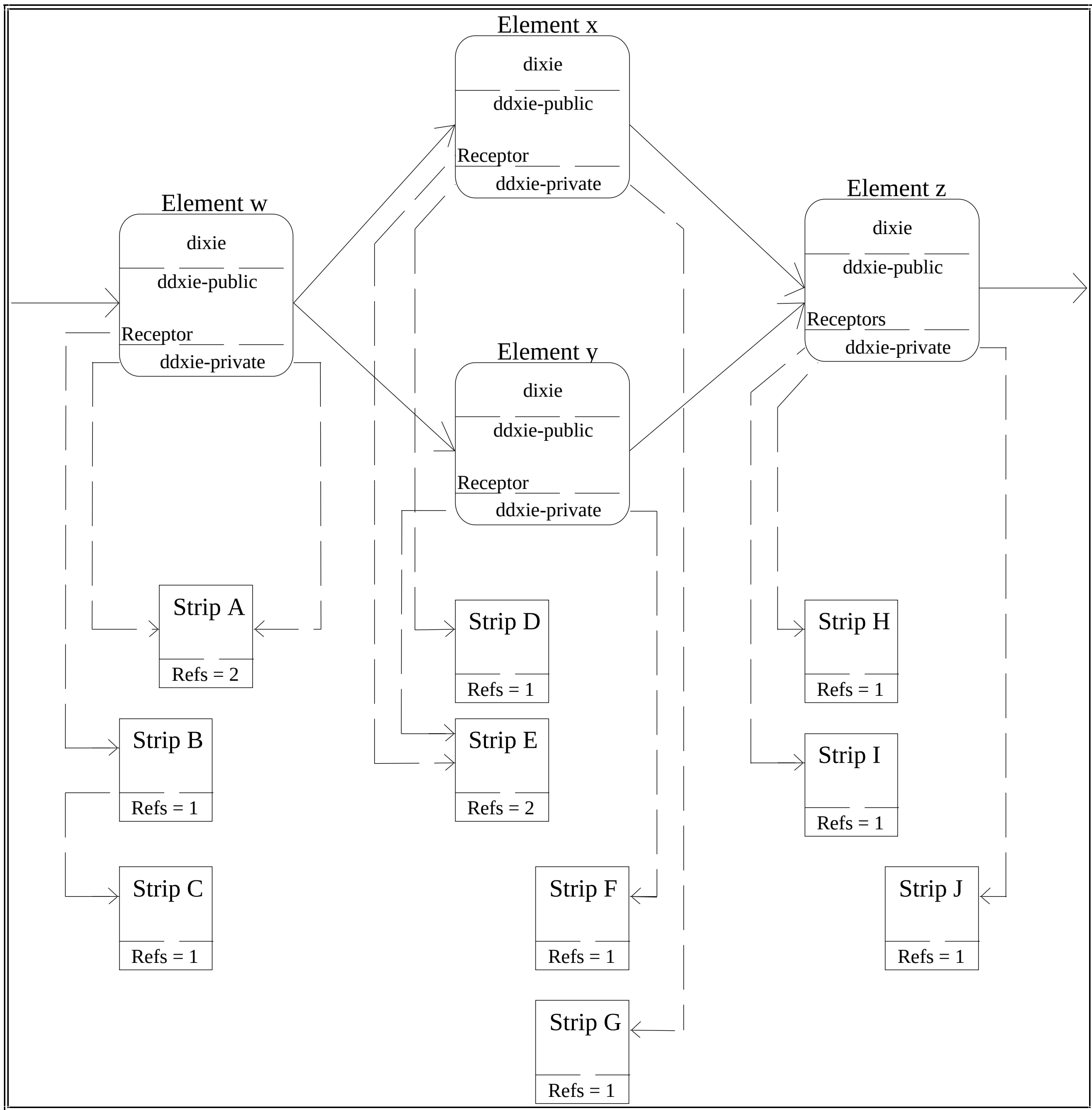
Elements access the data manager through five entry points:

- **MakeStrip** : used by an element to obtain storage for data it intends to produce.
- **PutStrip**: used by an element to give a completed strip of data back to the data manager (which then passes it on to downstream receptors)
- **GetStrip**: used by an element to "read in" a strip of data from its receptor
- **InPlaceStrip**: used to obtain in place privileges for a strip that it has obtained via **GetStrip**. If the data manager determines that another element also has access to the strip, a new strip is created instead.
- **FreeStrip**: used by a receptor to inform the data manager that it has no further need for the strip it obtained through **GetStrip**. Also used by an element to return an unfilled data strip buffer to the data manager if for any reason it decides to not fill it after allocating it through **MakeStrip** or **InPlaceStrip**.

To illustrate how data is passed from element to element, a runtime snapshot of an executing flo is shown on the following page.

In the illustration, element w is performing its processing in place with strip A being used both as an input and an output. Strips B and C are linked to element w's receptor indicating that they have not been read in yet. Since strip B is at the head of the list, it will be processed next. Any new strips assigned to element w's receptor would be linked after element C.

Strips D and E have been passed from element w to elements x and y. Element y has already processed the data from strip D and has indicated to the data manager that it has finished with the strip. Element y is currently processing Strip E and placing its output in strip F. Element x is currently processing strip D and has not yet read in strip E. Since element x hasn't read in strip E, it is still pointed to by element x's receptor. Element x is placing its result into strip G. Element z has two inputs. Strips H and I are being processed to form a result in strip J.



10.3.4. Input

Dixie calls through the flo-def input entry point when data arrives from the client via a PutClientData protocol request. Input calls upon the data manager to create a non- canonic strip that describes the byte stream segment, and then attaches it to the receptor of the import element handler that corresponds with the phototag specified to PutClientData. Input then passes control to the scheduler.

10.3.5. Output

Dixie calls through the flo-def output entry point when data is to be retrieved for the client. Output calls upon the data manager to fetch some non-canonic data from the export element that corresponds to the phototag specified in the GetClientData protocol request. If the client has asked for less data than is available from the export element, it will leave the leftovers with the export element. Otherwise, it will take all data that is available. Output returns the data and current output buffer status to dixie, which will send this information on to the client in the form of a reply.

10.3.6. Query

Dixie calls through the flo-def query entry point to find out if an import or export element handler is done. Dixie will interrogate each import and export element separately.

10.3.7. Abort

The photoflo will either complete normally, or dixie will call through the flo-def abort entry point to terminate execution prematurely. In either case the reset routine in each element is called. It will also call upon the data manager to release any strips it may have cached.

10.3.8. Destroy

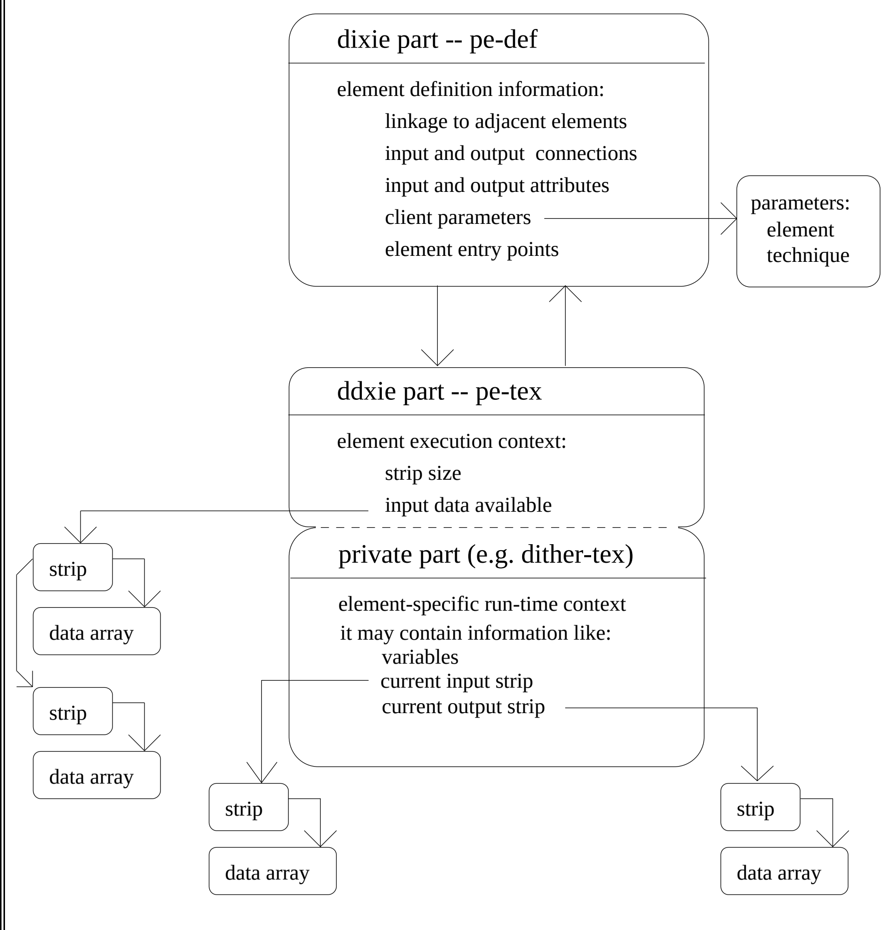
When dixie calls through the flo-def destroy entry point, the flo manager calls the destroy routine in each element and then frees the flo-tex.

11. Photo Elements

An element is the basic execution unit of a photoflo. An element consists of a body of code with supporting structures.

11.1. Element Structures

Element structures are separated into a dixie part, a ddxie part, and an element-specific part. The dixie photoflo management routines only understand the dixie part. The ddxie photoflo manager understands the dixie and ddxie public parts. Only the element itself understands the contents of the ddxie private part. These structures are illustrated below in conjunction with *strips* (which are handled by ddxie data manager routines and macros).



Element structures

11.2. Element Interfaces

The external interface to each element consists of a set of entry points that are loaded into the pe-def by the DAG analyzer and the photoflo linker. These entry points are used by various components of the photoflo manager for setup, execution, and shutdown of the element.

11.2.1. Create

The create entry point is called by the photoflo linker. It is responsible for attaching a pe-tex to the pe-def. It also loads the entry points into the pe-def. It will also allocate or setup anything that can persist over multiple executions.

11.2.2. Initialize

The initialize entry point is called at photoflo startup. It allocates and initializes run-time data for the current execution instance.

11.2.3. Activate

Activate is called by the scheduler when there is sufficient data available at one or more receptors. It calls upon the data manager to get data from its own receptors and to push data out to the receptors of downstream elements.

11.2.4. Flush

Flush is called when all of the upstream elements have completed their task. This gives the element one last opportunity to finish up.

11.2.5. Reset

Reset is called at photoflo done time (i.e. flo completed successfully, or aborted due to an error or client request). Reset frees run-time data and structures that are left behind by the initialize, activate, or flush routines.

11.2.6. Destroy

Destroy is called to free data and structures that are allocated by the create routine, including the run-time context itself (i.e. remove the pe-tex from the pe-def).