# Datacube MV200 and ImageFlow User's Guide

**Jim Vallino**
**University of Rochester**
**Department of Computer Science**
**June 1995**

## Table of Contents



## Introduction

The Datacube MaxVideo™ MV200 is a high performance image processing system integrated onto a single 6U VME circuit board. This is the third generation of hardware being produced by Datacube. The company has been building real-time image processing hardware since 1979. The MaxVideo systems are noteworthy for their ability to process video rate data. Their hardware was the first single board video rate image processing system available commercially. Traditionally, as might be expected, their customer base has been high performance military applications. Datacube has been courting industrial and medical applications the last year or two to expand this customer base.

The standard MV200 packs a lot of functionality onto one circuit board. It has:

- 24 Mbyte of memory in 6 banks,
- analog input section capable of 8 bit digitizing almost any input video format up to a 26 MHz pixel rate,
- analog output section capable of driving RS-170 monitors and multi-sync monitors up to 1kx1k resolution,
- 8x8 convolver,
- linear and non-linear ALU sections,
- statistics gathering hardware for histogramming or feature detection,
- binary morphology processor,
- 20 MHz cross-point switch for internal routing of signals through the hardware, and
- expandability via the on-board MaxModule port and the four external MaxBus ports.

Data primarily flows through the MV200 in entities called pipes at a rate of 20 MHz. All hardware processing elements operate at this rate. The memory modules also contain internal processing elements for simple arithmetic and statistical processing. Operations that can stay internal to a memory module are performed at a 40 MHz rate. It is expected that on the next major release of hardware Datacube will up all processing speeds to 40 MHz.

All of these high performance features do not come without a cost. Not only are Datacube systems expensive but they are particularly difficult to use. Programming of Datacube hardware has always been tedious at best. The software provided was obviously written by people who were mainly hardware engineers. The original MaxWare

software was little more than a high-level interface to registers directly controlling the hardware. This is hardly an interface appropriate for efficient programming of complex image processing algorithms. The latest ImageFlow™ software hides a large portion of those details from the user but, even with this improvement, the programmer's task is still rather tedious. It is compounded by notoriously poor documentation and absolutely no debugging support for this complex hardware. With all this summed up, Figure 1 is a familiar scene for most novice MV200 and ImageFlow users.

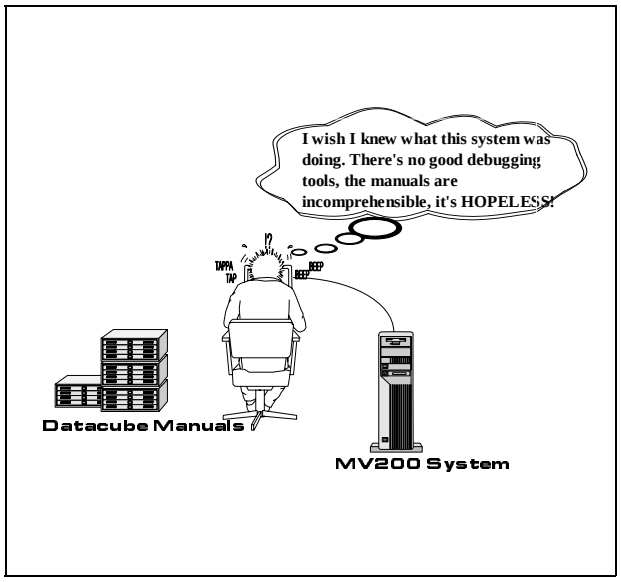


**Figure 1** The typical situation for a novice MV200/ImageFlow user

The MV200 is a high performance image processing system. It is best suited for applications with processing demands that surpass the capabilities of other easier to use systems. The traditional user community has been willing to live with the difficulty of using the Datacube hardware because it was the only system that could provide the performance needed. To have those capabilities the user must not be isolated from the hardware. Adding additional software interfaces, to ease the programmers burden, will almost necessarily have to isolate the programmer from some of the capabilities of the hardware. Users will quickly develop their own canned pieces of ImageFlow code for common operations. Datacube provides similar "pre-cooked" code with their SILL product. There also are other software systems that incorporate the MV200 as an element in a bigger system. Each application will have to assess what level of hardware access is needed to meet its functional requirements. The application may not need to program directly on the Datacube system. In this case, using the Datacube with another software system that makes access to the hardware easier would be acceptable.

## What this report will try to tell you

This report is intended as an introduction to the MV200 system and programming using ImageFlow software. It contains a description of the MV200 hardware, an introduction to the basics of ImageFlow programming, a guide to the documentation that describes both the hardware and software, and a short section on the physical interface to the hardware in the Robotics/Vision lab. (This last section is very brief because the information is subject to change whenever the lab is rearranged.)

There are several sample programs that highlight features on the MV200. The programs have been selected to show how some common image processing operations are performed on the MV200. Each tries to introduce an different aspect of programming the MV200 using ImageFlow software.

At the end of the report there is a glossary, instructions for running the MV200 demo program, a collection of some MV200 element flow diagrams and listings of the sample programs described in this report.

## What this report won't tell you

This report is not a description of all the hardware and capabilities of the MV200. It describes most of the features of ImageFlow programming but there are many sections of the hardware that are left uncovered. No description is provided for any of the statistical processing hardware, the morphology hardware, or how to work with the MV200 connected to external hardware such as the Digicolor board. There also is no description provided for using the ImageFlow graphics capability to provide, for instance, an on screen user interface like the one in the MV200 demo program. Hopefully, as the department community starts using the MV200 and uncovers how to make different parts and features work, additional reports will be written to disseminate that new knowledge to our user community.

## Once over the hardware and wiring

The MV200 is in the Robotics/vision lab. It is a single board plugged into one of the MaxVideo backplanes. The

hardware is connected to granite and you must be logged onto granite to use the MV200. (When you read this the system may have been moved to medusa with a second MV200 also installed there.)

Connectors on the edge of the board provide the input/output access. The lab staff has wired the video input and output sections to the patch panel in the lab. The input section has three video input jacks for composite video. The MV200 digitizes 8 bit gray scale. Color or black and white cameras can be connected to the input, but only monochrome signals will be processed by the standard MV200.

The MV200 output is on the patch panel on the row below the video input. It is red/green/blue/sync (RGBS) that can be connected directly to a monitor. There is no composite video output from the system. The output section of the MV200 can generate RS-170/NTSC output to drive standard monitors. If you are using the higher resolution output (up to 1k x 1k) make sure the output is connected to the high resolution multi-sync monitor. The MV200 will most often be connected to the later type of monitor. This will allow for display of multiple frames of video possibly showing intermediate results of processing. Another option is to include a user interface in the extra space on the output screen. Note that the multi-sync monitor can not sync to an RS-170 signal.

There are also four sets of external MaxBus connectors for connecting the MV200 to external hardware. These could be used, for instance, to attach a Digicolor board to the MV200 for processing of color video signals.

The MV200 itself is divided into several function parts called devices. The following is a brief summary of the devices that are in a standard system along with their two letter device designator:

- AB - the mother board that ties together all of the hardware. The major component is a 20 MHz cross point switch for connecting anything to anything. The memory devices are associated with the AB.

- AS - **A**nalog **S**canner. An incoming video signal is digitized here.

- AG - **A**nalog **G**enerator. This device generates the video output in one of several formats.

- AM - This is the memory device which holds the main memory used for storing images. A standard MV200 has 6 banks of 4 Mbyte multi-ported memories. Each memory bank also has local ALU and statistics calculating elements that run at 40 MHz for operations within one memory bank.

- AU - **A**rithmetic **U**nit. The linear and non-linear ALU hardware is on this device.

- AP - **A**dvanced **P**rocessor. This device contains the more advanced processing elements. Among these are: a statistics element, neighborhood multiple and accumulator element, and look-up table elements. There is also a MaxModule connector for adding additional hardware to the system.

To get a flavor for the capabilities of the MV200 it would be worthwhile to run the demo program. Look at Appendix A for a step by step guide on how to run the MV200 demo program.

## The manuals - you'll need these!

If you are someone who does not like to read the manuals then working in the MV200/ImageFlow environment will be very challenging. The level of detail required for programming this system necessitates a good familiarity with the Datacube manuals. Most of the information needed is contained in the manuals. It can sometimes be rather cryptic and, unfortunately, there still is some folklore surrounding the Datacube system that can only be learned with trial and error.

There are four manuals provided with the system. Two of these will be used as constant references when writing ImageFlow programs.

***Datacube IP Manual***. This manual provides the introductory information about the hardware and pipelined image processing. It describes all of the generic hardware elements that are in the MV200 and introduces the terminology and symbols used throughout the rest of the manuals. This manual should be reviewed first before proceeding to the other manuals.

***ImageFlow System***. This manual gives an introduction to the ImageFlow software system. The programming conventions and standard nomenclature for describing ImageFlow programs and functions are specified here. There is a section with example programs that are of questionable value since they are not written for the MV200. After an initial reading, this manual is mostly referred to for the sections on Event Manager, Error

Handler, Graphics, or General Input. Handling events is an important part of ImageFlow programming that is discussed toward the end of this guide. The graphics and input section give details needed for merging a user interface with the output generated by the MV200.

***Hardware Reference Manual.*** This manual gives all the details about the hardware capabilities of the MV200. There are sections for each hardware device. Each device has an Installation and Specifications chapter and a Functional Description and Usage Guide chapter. The first is rarely used after installation of the hardware. The Functional Description describes what features and capabilities each device has. A thorough knowledge of these descriptions is needed to fully understand the capabilities of the MV200 hardware.

The first item to look at for a particular device is the element flow diagram. This diagram (there may be several pages of them for a device) shows a schematic of the device. The element flow diagrams for some MV200 devices are in Appendix A. It will be helpful, at least initially, to make copies of these diagrams so that you can mark up the data flow through each device in your program. Each element shown on the flow diagram is given a name prefaced with the two character device designator. The names are meant to be descriptive but it takes a little time for this descriptiveness to become apparent. For a moment take a look at the AG diagram. Data flows are normally from top to bottom. Single weight lines are assumed to be 8 bit paths unless they are labeled differently, such as the output of AG_ALU_AND that is 4 bits wide. Bus combining and splitting occurs frequently. The output of the AG_DAC_LUT is three byte wide signals that are bussed together to form a 24 bit path. The byte wide device input DQ_CGO (upper right corner of diagram) is split into two 4 bit paths. The ordering of the data is always strictly maintained as most significant to least significant from left to right. The creation of the AG_8BIT and AG_15BIT signals is another good example of the bit shuffling that is done. Try to sort out what is being done here based on the descriptions of the signals given in the AG functional description.

The next part of the functional description to look at is the Detailed Functional Description. It lists each element that is on the device. Elements of a given type have different features and options as described in the *Datacube IP Manual*. Elements of the same type can have different capabilities within one device or between elements on different devices. This section describes the capabilities for the particular elements on this device. You will need the information about each element to know how to set it up. Additional descriptive text is given for any element or device feature that may require further clarification. The last section of the functional description is a listing of the standard initialization state. Information in this section can be used to reduce the amount of programming necessary. If an element is initialized to the state that is required the programmer does not have to deal with that element in the program. (It may be better practice, though, and make for easier to understand code, to not leave anything to chance and explicitly specify all settings.)

***ImageFlow Reference Manual***. This manual provides documentation for all of the ImageFlow functions. This is a large library of C callable functions that provides programmer access to the system. Most functions have a direct effect on parts of the hardware in the MV200 system. Other functions pertain to ImageFlow management of the system, and the facilities for drawing graphics and obtaining user input. The documentation follows the standard Unix™ man page format with the functions listed alphabetically. There is a *Quick Reference Guide* that gives the synopsis information for all of the functions. This can be useful when you can't quite remember the name of a particular function. The index at the end of this manual is also helpful. It is indexed by operation. If you need to perform a certain operation, this index can point to the functions that perform that operation on different elements. Finally, the Tools/Utilities Appendix section describes several features and programs that are occasionally used.

## ImageFlow ABC's

The basic terminology and nomenclature used in ImageFlow programs will be described in this section. Both the *Datacube IP Manual* and *ImageFlow System Manual* should be read for a more thorough and exhaustive description of the Datacube hardware and ImageFlow software models. There is a Glossary of the important terms at the end of this report that may suffice as a quick reference to the terminology.

As mentioned in previous sections, the MV200 system is composed of several devices with each device containing multiple hardware elements on it. Datacube defines generic hardware element types to perform certain operations. On each device these generic elements are combined in a particular fashion to provide the capabilities of the device, such as video input or statistical processing.

Images, or in a more general sense, data flows through the MV200 in ImageFlow entities called pipes. Datacube refers to these entities as objects, but ImageFlow provides only the very rudimentary underpinnings for an object oriented approach to programming an MV200. Significant work would be required to define object structures and classes to follow a true object oriented paradigm. (Mike Swain and his colleagues have written a class structure for working with the MV200 in the context of a Datacube server. The author also has several C++ classes defined for working with ImageFlow.) At the start of every ImageFlow program the program must identify the devices that will be used and get an ImageFlow handle to them for future reference.

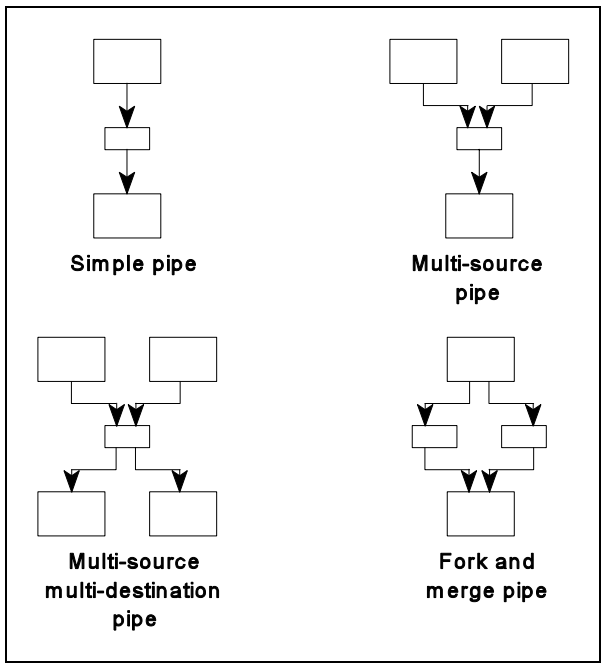


**Figure 2** ImageFlow pipe topologies

An ImageFlow pipe can have several different topologies as shown in Figure 2. The pipe internally may route through several devices with elements on each device performing operations on the data. The ImageFlow program defines how the different elements are configured to make the pipe. There can be several pipes operating at the same time on an MV200. A typical program has three pipes: acquisition, processing, display. The acquisition pipe acquires frames of video data using the AS device. This data is stored in AM memory elements to be processed through the processing pipe. The result of these operations is displayed on an output monitor by the display pipe. These three pipes can run independently or

ImageFlow has the capabilities to synchronize the pipes using its event manager. The basic steps to use a pipe are:

1. define all the connections and element parameters in the pipe,

2. create the pipe as a single or multi-destination pipe,

3. arm the pipe,

4. fire the pipe to get it running.

More details of these operations will be given in the discussion of the example programs.

A pipe can be configured, at the time when it is created, to run continuously or only once (one-shot). Depending on the problem requirements, both modes of operation are regularly used.

*Hint 1*
*The display pipe should always run in continuous mode to get a stable output display on the monitor.*

There can be up to four pipes running continuously. This limitation is imposed by the number of available timing signals on the hardware. There are methods to handle processing tasks that can not stay within the four pipe limitation. The most common method is to predefine the different pipe configurations and switch between these configurations. The switch can be accomplished with little overhead during the actual running of the program using Pipeline Altering Threads (PATs). This technique is discussed in the examples.

Data is stored in the MV200 in objects called memory surfaces. Many elements can have surfaces defined on them. The most common elements in which surfaces are defined are the AM memory elements that usually store image pixel data. There are six 4 Mbyte multi-ported memories. A memory surface is a two dimensional array of data. Variable format memories, such as the AM devices, allow the user to define the dimensions of the surface.

*Hint 2*
*All branches of pipes should begin and end at a surface. There should never be a surface in the middle of a pipe.*

Surfaces are defined in other more unusual locations also. For instance, digitized video data is accessed by declaring a surface on the A-to-D convertor in the AS device. Defining this surface partially specifies the parameters for digitizing the input video signal. The AS device can digitize a wide range of input signals. Similarly, a surface must be declared on the AG device's output D-to-A. The dimensions of the surface specify the output display format that will be used. The AG output section can generate not only RS-170/NTSC video but it can also drive multi-sync monitors up to a 1k x 1k resolution. The AG device can not be configured to output arbitrary sized surfaces. The choices are fixed by the hardware.

*Hint 3*
*The aspect ratio of the image will change when displayed on a monitor that does not preserve the aspect ratio of the input pixels. 4:3 pixels from an RS170/NTSC signal sampled at 512x484 will look strange when displayed on a square pixel monitor.*

Surfaces are also defined in the neighborhood processor - to define the convolution kernel and in the statistical elements - to define the location to store results of statistical operations. The programmer has access to the various surfaces that were defined through ImageFlow functions.

The following sections of this report discuss in more detail specifics of using ImageFlow to program the MV200. Each section will introduce new aspects of using the hardware and software. The collection of all these programs does not come close to exhausting the capabilities of the MV200 or ImageFlow. The intent, rather, is to give the reader enough background knowledge so that they can explore the other features of the system when it is needed to solve an image processing problem.

A listing of each example program is in Appendix C. The source files are also in /u/vallino/Datacube/guide/src. Executables are in .../guide/bin. There is a README file there to remind you of the required setting for environment variables, path, etc., to run the program.

## A simple flow using ImageFlow

This program performs the most minimal processing that can be done. It digitizes a video signal in the AS device and stores these frames in an AM memory. This same memory is then used in the display pipe so that the video can be displayed. Figure 3 shows the simplified flow diagram for this program. It will be helpful to follow this description with the pertinent element flow diagrams and the *Hardware Reference Manual*.

The first sections of this file, simple.c, does the standard initializations that must be performed in all ImageFlow programs. The file *datacube.h* must be included. This file defines all of the data types, constants and functions that are part of ImageFlow. All of the devices on the MV200 that will be used in this program must be specified in the *dqLimitIPDevSet()* statement.

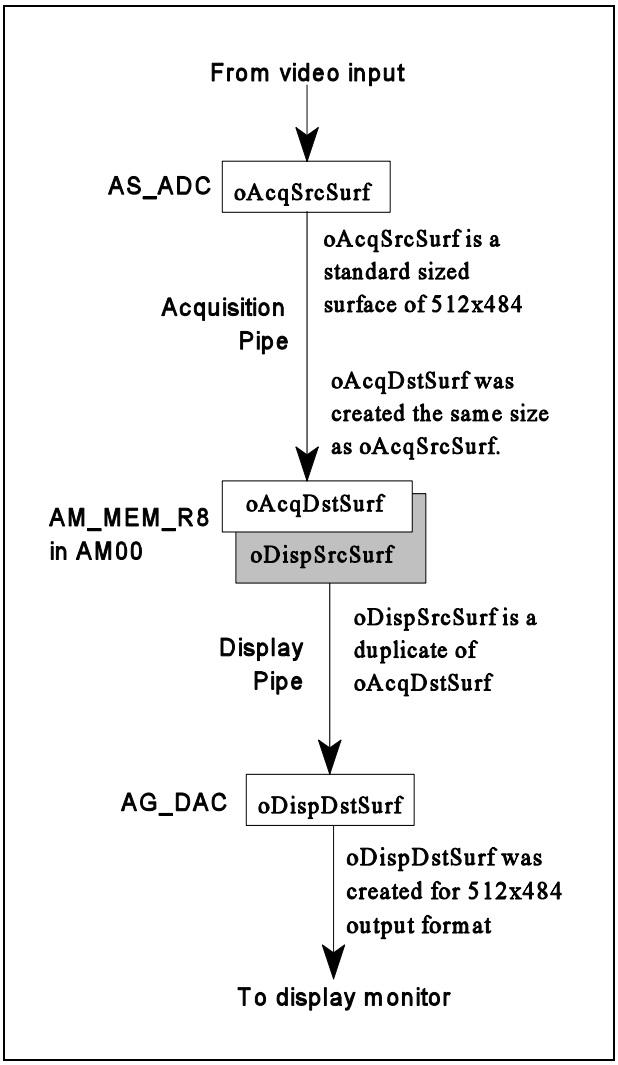


**Figure 3** Pipe diagram for simple.c

Most of the objects used in ImageFlow have specific data types defined for them like DqIPDev for devices or DqSurf for surfaces. The ImageFlow environment must be

initialized with *dqInitEnv()*. The next step in the initialization process is to get handles for the devices. First, a handle for the whole system must be gotten with a call to *dqCreateStdSys()*. This function reads the standard configuration file to determine the hardware configuration. The configuration file to use is specified by the environment variable DQCONFIG. If this is not defined then the standard configuration file, dqsys.cfg, found under the $DQHOME directory is used. The standard configuration file specifies stand alone operation of the MV200. If you are using the MV200 with other hardware, such as an external Digicolor board, then you will have to specify a configuration file that includes that other hardware. The call to *dqCreateStdSys()* will verify that all of the hardware is connected as specified in the configuration file. There are many other parameters in the configuration file. The System Configuration and Set Up section of the *ImageFlow System Manual* gives a full description of configuration file.

The last initialization step gets ImageFlow handles for the devices that the program will use. All these devices must have been defined in the *dqLimitIPDevSet()* statement at the top of the program. The name for the device must match the name in the configuration file. For an MV200 only the AB device is listed in the configuration file. All of the other devices are implicitly associated with it. Note that the AM memory devices are referenced differently than the other devices. The configuration file will allow more than one MV200, i.e. AB device, to be in the system. This might be needed for very high performance image processing tasks. In this case, the second MV200 might be called AB01 and 01 would also be used to designate all of its devices.

The acquisition pipe is defined next. The standard initialization of the AS device is for RS-170/NTSC video. No adjustments of input timing parameters are needed if this format signal is used. The memory surface that begins the pipe is defined on AS_ADC. The RS-170 512x484 format is the standard sized surface. You find this listed in the Standard Initialization State section of the AS Functional Description and Usage Guide in the *Hardware Reference Manual*. (Got it?) Also, note that the AS_SRC multiplexer's data source is initially set to AS_V0. This program will digitize data from that input channel. It is user preference whether that connection, or any other default, should be explicitly specified in the program code. To get anything into or out of a memory you must attach a gateway to the surface. There are transmitter and receiver gateways for writing into and reading from memory surfaces, respectively. The program attaches the transmit gateway to the surface just created.

The hardware that ties all of the MV200 devices together is shown on the AB device element flow diagram. The big multiplexer that spans the entire width of the diagram is the 20 MHz cross point switch. It is very difficult to do anything without having to connect through the cross point switch one or more times in a pipe. Any input to the switch can be connected to any number of outputs simultaneously. Only one signal can be connected to an output. The *dqConnect()* function is used for all connection of elements on the MV200.

Looking at the cross point output AB_OP00 you will notice that it is directly connected to DQ_IMR0 which supposedly goes to AM00. If you look on the AM element flow diagram there is no similarly labeled input signal. There are several implicit connections between the AB device and the 6 AM devices. These are specified in the section 2.3.3 - AB Device Connections to the AM Device. The matching input, in this case, is DQ_CM0 and that is where the signal will appear. (Datacube was never known for producing clear documentation!)

*Hint 4*
*Signals sometimes cross between devices and magically are no longer referenced the way you expect on the element flow diagrams.*

The AM memory devices operate internally at 40 MHz. When it will be accepting data from a pipe, it must run externally at the 20 MHz speed of the rest of the system. This speed match up is done with a call to *amSetRcvGateway20MHz()*. You can include this code in each of your programs or create a separate file for this and several other auxiliary functions that are needed to more easily program the AM gateways. The surface created in AM00 will receive the video data digitized in the AS device. As shown on the AM element flow diagram, there are several memory elements defined. Each one represents a different bit resolution for the data stored in the memory. Most often the AM_MEM_R8 element will be used to store byte data. A receive gateway is attached to this surface after it is defined.

All of the connections in the acquisition pipe are specified at this point. The next three statements perform the steps necessary to finish building and starting the pipe. First, the pipe is created and a handle for it obtained with a call to

*dqCreatePipe()*. This will create a single destination pipe. There can be multiple sources in a single destination pipe, but in this case there is only one. The surface that terminates the pipe is specified as the first parameter. The pipe will run continuously once it is started. One firing of the pipe entails processing all of the data from the source surfaces through the pipe into the destination surfaces. When a transfer is completed, a continuously triggered pipe will start another transfer. *dqHaltPipe()* will stop a continuously running pipe if needed. For the acquisition pipe one transfer represents digitizing one frame of video data and storing it in oAcqDstSurf. The hardware handles all of the details of interlacing on the input video signal. Odd and even fields are correctly merged into one frame in the first memory surface in which they are stored.

*Hint 5*
*A safe policy to follow is to place data from an interlaced video source in a memory surface right after digitization. It is possible to process data directly out of the AS device. This will give unexpected results for neighborhood processing however.*

Pipes can also be created as one shot pipes with the define DQ_TRG_ONESHOT. Each time the pipe is fired it will perform one transfer and then stop.

*Hint 6*
*The triggering mode can not be changed after the pipe is created. To change it the pipe must be destroyed and rebuilt.*

Before a pipe can be fired to get it running a call must be made to *dqArmPipe()*. The create pipe operation does little more than some rudimentary checks and returns a handle for the pipe. The time consuming part of building the pipe is performed when the pipe is armed. The second parameter in the call tells ImageFlow how much has changed in the pipe since the last time it was armed. There are several categories of changes defined on the man page for this function. Whenever the topology of a pipe has changed, either by being newly created or if connections are changed later, the most time consuming operation, DQ_DSM_PIPE, must be specified. This will recalculate all delays and connections through the pipe. (More will be said about this parameter in a later example that changes the pipe after it is initially created.) Finally the pipe is fired. The oAcqDstSurf is now being continuously updated with the data from the input video.

The second part of the program builds the display pipe. The digitized frames of video that we want to display are being stored in oAcqDstSurf by the acquisition pipe. To get access to this data, the display pipe uses an ImageFlow feature that allows the programmer to declare multiple surfaces in a memory element. Each time that a *dqCreatexxxSurf()* call is made new memory in the element is allocated to the new surface. This memory is not shared with any other surfaces. Surfaces can be created until the available memory in the element is exhausted. The size of a new surface can be specified directly with *dqCreateSurf()*, default to the standard size for the element with *dqCreateStdSizeSurf()*, or be created the same size as another surface with *dqCreateSameSizeSurf()*. None of these calls, however, do what is needed here, namely give access to the same memory allocated for another surface. This is accomplished with the *dqDupSurf()* function. The surface created by this call references the same memory that is being written into oAcqDstSurf by the acquisition pipe.

*Hint 7*
*A memory can have many separate surfaces defined in it but the receive and transmit gateways can each attach to only one surface at a time.*

The AM memories are triple ported devices. There are ports into the memories from the receive and transmit gateways. The host computer also has transparent access to the memories via the VME bus. Host access can occur while the other two ports are active.

The outputs of the first four AM memories are connected (via another set of implicit AB board connections) not only to the cross point switch but also directly to the AG device. These four memories can be used for displaying output images. The first three can display normal pixel data and the fourth one is reserved for display of overlay data. To allow the AG device to refresh high resolution monitors the connection between these memories and the AG device can run at 40 MHz. Similar to the input of the AM memories, the external speed of the memory must match what it will drive. If the output of the memory is only used in a display pipe it should output data at 40 MHz to allow display at all output resolutions. If the data is used as part of a processing pipe, then the memory must output at 20 MHz to match the speed of the cross point switch. It is

possible for a memory to be used for display and processing at the same time if it outputs data at the 20 MHz rate with the limitation that only monitors with an effective pixel rate of 20 MHz or less can be used. simple.c uses the auxiliary function *amSetDispGateway40MHz()* to set the memory output speed in the display pipe.

There is another speed matching that must be performed when displaying output with the AG. The AG_RCV receive gateway on the AG device always samples and outputs at 40 MHz. If the display format does not have an effective data rate that is this high, the AG receive gateway must be fooled into thinking that it is handling a data stream with a 40 MHz data rate. This is done by controlling the transmit expansion on the memory transmit gateway and the receive shrinkage on the AG receive gateway. Two items determine the appropriate multipliers to use. First, is the effective pixel rate for the display format being used. The display format is set by the size of the surface created in the AG_DAC element. The formats supported are listed in Table 2-2(AG) of the AG Functional Description and User Guide in the *Hardware Reference Manual*. Find the effective data rate for the output format that you will use and then move to Table 2-3(AG) to determine the correct horizontal expansion and shrinkage factors to use. simple.c is displaying in RS-170 interlaced format which is specified by the 512x484 surface size when the oDispDstSurf is created. This format has an effective pixel rate of 10 MHz. Table 2-3(AG) specifies to set both the horizontal expansion and shrinkage to 4. This is done with the two calls *dqSpecXmtExpansion()* and *dqSpecRcvShrinkage()*.

There are two other straight forward connections to make on the AG device before the surface is created and attached to its receive gateway. Finally, the pipe is created, armed and fired. If a video source is feeding the AS_V0 input and an RS-170 monitor is attached to the output a live video image should be seen on the monitor.

There is a make file in the .../guide/src directory. This can be used as a template for compiling your own programs. You should also define the environment variable DQHOME to /s8/imageflow in your .cshrc file.

*Hint 8*
*Datacube executables are HUGE! The executable file for simple.c is over 600 Kbytes! Please keep your directories clean by deleting unneeded executables.*

## Add a convolution

This example will perform a 3x3 convolution, at video rates, on the digitized video signal. The convolution is performed on the AP device in the neighborhood multiply-accumulate (NMAC) element. This program is in the file convolve.c. A skeleton flow diagram is shown in Figure 4.

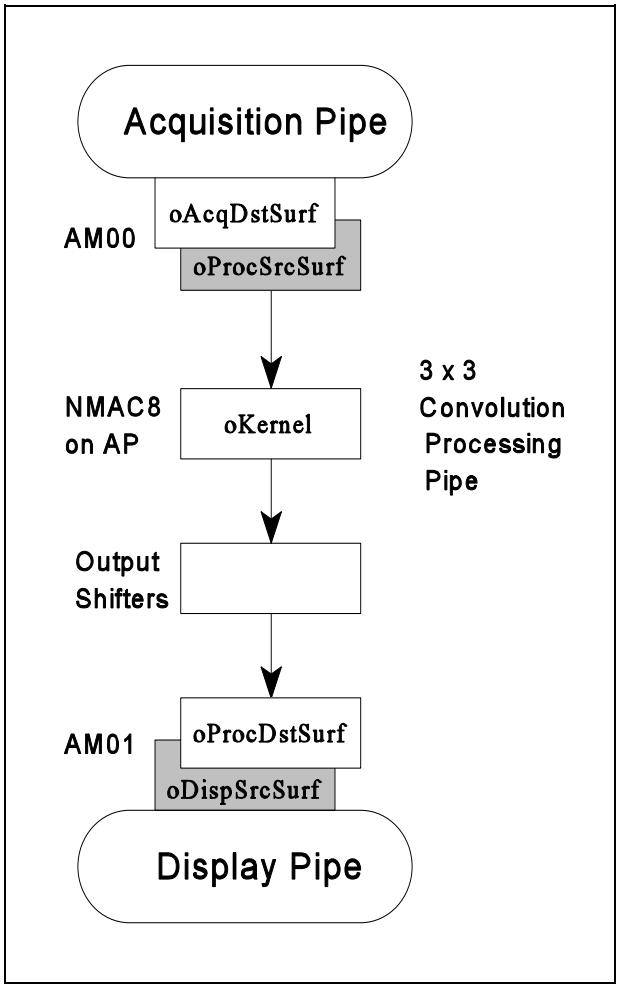


**Figure 4** Pipe diagram for convolve.c

Looking at the program listing, notice that the AP device is declared at the top of the file and a handle is gotten for it. A second memory device, AM01, is also used in this program. Our convolution kernel will have negative coefficients and we need to have the data interpreted as signed data. The ImageFlow software matches data types in pipes by transparently setting conversions in elements. To allow for this transparent matching of data types the programmer must set type checking to tolerant using the

ImageFlow function call *dqSetDTPDogma(*DQ_DTP_TOLERANT*)*, otherwise, data types will be mismatched in the pipe. The default data type is unsigned byte. If you are using another data type be sure to specify it for the appropriate elements in the pipe.

*Hint 9*
*Don't bother being compulsive and requiring the programmer to explicitly specify all data types using the DQ_DTP_STRICT dogma. That feature does not work and will be removed from a future ImageFlow version.*

The acquisition and display pipes are basically the same configuration as in the first example. This discussion will highlight the processing pipe. The source for the pipe is a duplicate of surface oAcqDstSurf. This data is going to the cross point switch so the memory output must be set to operate at 20 MHz using the auxiliary function *amSetXmtGateway20MHz()*.

Many of the MV200 devices can have sections configured differently on a larger scale than just setting an operating parameter or mode. These configurations are highlighted on the element flow diagrams in gray blocks. For example, the NMAC on the AP device can be configured to perform either an 8x8 or dual 8x4 convolution. This program connects AP_SHIFT8 to the AP_NDLY_SRC to select the 8x8 convolution mode.

*Hint 10*
*Gray configuration blocks are chosen by selecting the data source for the output multiplexer that is connected in the configuration that you want.*

The convolution kernel is defined as a surface in the NMAC. This program will perform a 3x3 convolution to detect horizontal lines. The surface is created in the usual fashion. Note that the data type for the surface must be set before writing into it so that the negative coefficients are handled properly. The next step is to write the coefficients into the kernel surface. This is a multi-step process.

First, a we must define the rectangle that we will write to on the kernel surface. All surfaces have a default zero point. This is called the alignment point of the surface. (More will be said about alignment points in normal surfaces later.) The default zero point for the kernel is the middle of the kernel. If the kernel dimensions are odd then this is the center point in the kernel. For even dimensioned kernels it will fall between two pixels. During processing the center point of the kernel will align with the center pixel of the neighborhood being convolved. (Datacube has never been very clear about how between pixel alignment operates. The limited tests I have done indicate that alignment is to the nearest pixel.) This program will leave the alignment at the center value and defines the rectangle from (-1,-1) to (1,1).

*Hint 11*
*The alignment point does not have to be placed within the kernel surface. This might be done for instance when composing a larger kernel from several passes with smaller ones.*

The next step is to calculate the coefficients for the kernel. This requires that we take into account the data path through the NMAC and AP device. We will be sending byte wide data into the convolution process and want to get byte wide data as the result. The three high bytes of the 40 bit output from the AP_FORMAT element are routed to the cross point switch. We can select one of those three bytes for the output provided that the convolution and subsequent processing puts our output data in the correct bit positions. We will use the high order byte, DQ_CP15 for output. Now we must determine what implications that choice has on the setting of coefficients and shift values.

AP_NMAC8 is capable of performing an 8x8 convolution with 8 bit kernel coefficients on 8 bit data. The convolution can be less than the 8x8 size. (Larger convolutions can be composed using the standard technique of multiple passes.) When you choose the size and coefficients for your kernel calculate the bit size for the output maximum sum. The convolution kernel in this program uses values of 64 and -64. The input pixels are signed values with a maximum of 127. The maximum positive output using the kernel in this program is 64 * 127 * 3 = 24,384 = 0x5f40h. The maximum negative output is -64 * 127 * 3 = -24,384 = 0xa0c0h. This convolution require 16 bits to represent the maximum output value. The value does not change when the data passes through AP_ADD because we have set the three high order bytes for the AP_ADD B operand to zero. This makes the add operation an identity operation.

*Hint 12*
*Each output of the cross point switch can be connected to a constant. The "connection" to a constant is made by using the output name as if it was a constant element in the dqSetKVal() function.*

Since only the high 3 bytes of the 40 bit data path is available at the output of AP_FORMAT. With no shifting of the data, the default initialization, our data is not available at the output. This program uses the high byte output and requires a total of 24 bits of shifting to get the most significant byte of the 16 bit output of the convolution into the proper place.

*Hint 13*
*The data flowing through this pipe is signed. Be sure that you take into account the sign bit when calculating the shift values. You do not want to shift your most significant bit of data into the sign bit position. If the shifter detects a value change in the most significant bit, indicating a sign change, it forces the output to the maximum positive or negative value.*

You can give or take bits in the coefficients or in shift values until the desired result appears in the proper place. With that determined the coefficients can be written by the host into the kernel surface with the call to *dqWtRect()*. The proper shift values in the two shifter elements in the data path are also set. Because edges are represented by convolution output values that are tending toward the maximum positive and negative output, AP_FORMAT is set to AP_FMT_ABSOLUTE (2's complement of a negative value and 1 bit left shift of all values) to convert the data back into 8 bit unsigned values.

This pipe is finally created, armed and fired in the standard manner. After the display pipe is created the result of the convolution operation should be visible on the output screen.

## But not over everything

The last example program made reference to the alignment point of a surface. In that program the alignment point specified how the convolution kernel would align with the pixel data. The alignment point and another concept called the processing rectangle are important for controlling the processing of data in memory surfaces.

One of the major improvements of the later releases of the ImageFlow software is that the user is relieved of most of the requirements to calculate delays through the pipes. ImageFlow does these calculations to ensure that pixels in the source surfaces are processed and the results deposited in the correct pixels of the destination surfaces. It was particularly difficult to determine these delays when multiple surfaces were being merged in a pipe. All settings for internal delay elements are calculated during the call to *dqArmPipe()*.

The user can control which pixels in surfaces align during processing. Figure 5 shows the interaction between setting the alignment point and processing rectangle for two surfaces. Delays in individual branches of a pipe will be calculated so that the alignment points for all surfaces in that pipe match up. A convenient image to conjure is that all of the alignment points are nailed together. The default alignment point for a surface is set to (0,0) when the surface is created. A surface's alignment point is adjusted using *dqSpecSutfAlginPoint()*.

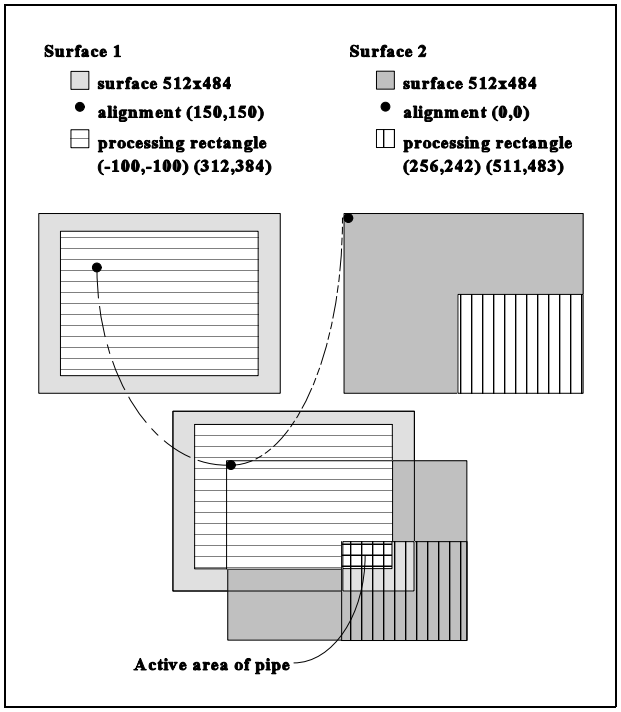


**Figure 5** Alignment point and processing rectangle = active area

The program can also specify a rectangle within a surface for active processing. This rectangle is called the processing rectangle for the surface. Only data that is within the processing rectangle will be processed in a pipe. The default processing rectangle for a surface is the entire area of the surface. A surface can have one alignment point and one processing rectangle specified. These are easily changed after the pipe is created to modify processing on the surface. Initially, the duplicate surface inherits the alignment point and processing rectangle of the original surface. The two surfaces are independent, however, so that the alignment points and processing rectangles of both surfaces can later be set to different values.

The alignment point is specified as an absolute (X,Y) position in the surface. It can have negative values. The processing rectangle is specified by its upper left and lower right coordinates relative to the current alignment point. These are inclusive coordinates for the rectangle. The processing rectangle can have negative coordinates if the current alignment point is in the middle of the surface. An error occurs at run time if any part of the processing rectangle extends outside the area of its surface. The processing rectangle is specified relative to the alignment point but it is stored as absolute pixel positions in the surface. If you change the alignment point the processing rectangle does not adjust.

*Hint 14*
*Always set the processing rectangle after any adjustment to the alignment point.*

To determine the area of pixels that will actually be processed, first, the alignment points of the source surfaces are "nailed" to the alignment point of the destination surfaces. The region of pixels processed is the intersection of the processing rectangles from all the aligned surfaces in the pipe. A run time error will occur if the resulting active region has zero area. Figure 5 shows an example of this determination of the active region. Only the small double hatched area will be processed. Assume that surface 1 is the source for the pipe and surface 2 the destination. The pixels coming from the lower left corner of the processing rectangle on surface 1 will be processed and the resulting values will be placed in the upper left corner of the processing rectangle of surface 2. The other pixels in surface 2 will remain unchanged. Alignment point and processing rectangle give the programmer large amounts of control over the placement and processing of data in surfaces.

*Hint 15*
*Virtual stores such as the AS_ADC and the AG_DAC do not operate the same as memory stores for alignment point and processing rectangles. Attempting to set an alignment point or processing rectangle in these surfaces is not very productive.*

The example program, procrect.c, is the convolution program modified to allow the user to specify a processing rectangle and alignment point for the destination surface of the processing pipe.. The contents of the surface outside the processing rectangle is unchanged. When a program makes use of processing rectangles the programmer will have to set the alignment point and processing rectangle at unexpected locations. When in doubt set them! Also, notice that the when the processing pipe was armed after setting the processing rectangle and alignment point, the DQ_DSM_DELAY change hint was given instead of the DQ_DSM_PIPE value used previously. This will shorten ImageFlow's time to process the arm command and is permitted because we did not change the pipe's topology. When in doubt use DQ_DSM_PIPE.

*Hint 16*
*The ImageFlow Data Stream Manager will implicitly adjust processing rectangles on destination surfaces when the pipe is armed as described in section 6.6 of the ImageFlow System manual. This may result in a smaller processing rectangle being set without the programmer explicitly specifying to do so. Use the dqInqProcRect() function to check when you suspect a problem.*

## Some other spiffy math

This example program demonstrates using the linear section of the AU device to perform simple arithmetic operations in a pipe. There is also a non-linear section of this device that can perform logical operations on pixel data. This hardware is not used in this program.

average.c is an ImageFlow program that executes a recursive time averaging filter on a video data stream. The filter implemented is:

$$y(n) = K*x(n) + (1-K)*y(n-1)$$

where: y(n) = the averaged video frame
y(n-1) = the last averaged video frame
x(n) = the new incoming video frame
K = a constant between 0.0 and 1.0

A flow diagram for the processing pipe in average.c is shown in Figure 6. The AU has an input section that can combine 8 bit data paths to do 16 bit arithmetic. All pixel data gets in and out of the linear section through the gray scale cross point switch. The linear section of the AU is shaped like an inverted tree. There are four data extenders at the top followed by multipliers.

*Hint 17*
*The data extenders, constants and ALU's are sensitive to data type. To avoid any unexpected results be sure to explicitly specify if signed or unsigned data is being used.*

Sixteen bit arithmetic is accomplished in four parts that are then combined for the final output result. This program using fixed point binary in the calculations to allow multiplications by the fractional value K. The binary point is set between bit 7 and 8. Determination of the constant and shift values takes this into account.

*Hint 18*
*Signal widths increase and decrease seemingly at random in the linear ALU section. Bits are often added and dropped in the data flow. A careful tracing of the flow is necessary to assure that the proper calculation is performed.*

Figure 6 shows how the complete calculation of the time averaging filter is built through the linear arithmetic section.

The shift performed in AU_L_SHIFT3 scales the data back down to byte range. The shift amount is set only to 7 bits even though the calculations are performed with a 8 bit fixed point fraction. The reason for this is that the output of AU_L_ADD3 drops the least significant bit from the data stream. The program adds a 1 at this point to effect a rounding of the result before the lowest bit is lost.

The linear section of the AU has many other elements for clipping, shifting, absolute value and adjusting the result of the calculation. Data leaves the AU through an output section, after passing once more through the grey scale cross point switch.

Try several different values for the averaging constant to see the effect on the displayed image.

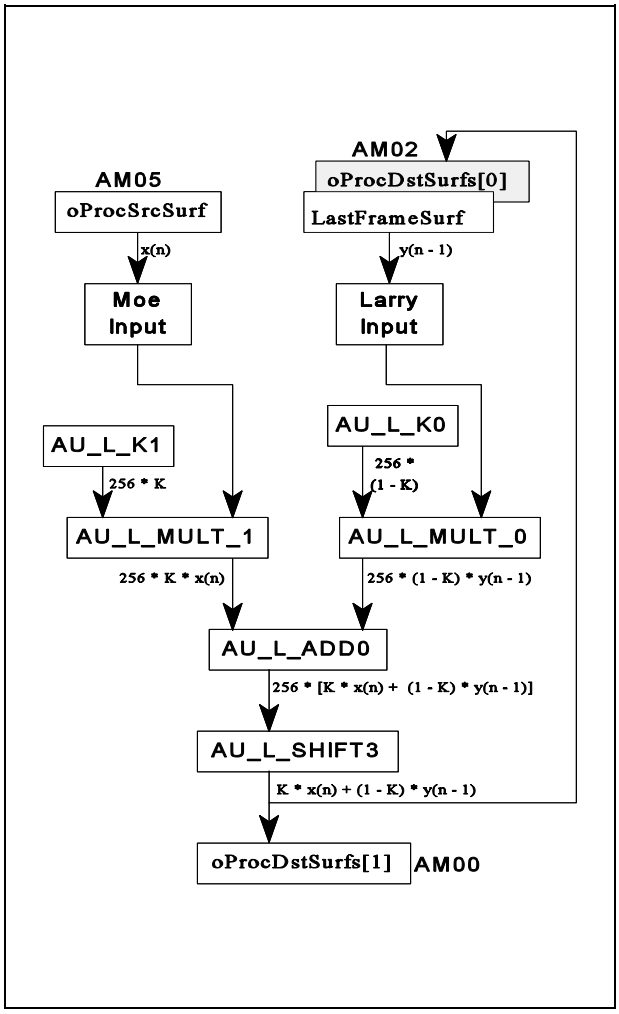


**Figure 6** Processing pipe in average.c

## MOSC is not the religious place

We have seen previously how to specify which sections of a surface are processed using the alignment point and processing rectangles. This control has the granularity of an entire surface and is limited to control of regions that are rectangular in shape. The processing rectangle also completely shuts off processing of pixels outside of the active rectangle. This program will show how to apply the

time averaging filter from the last example program to only a portion of the image while leaving the rest of the image as live video with no time averaging. Setting a processing rectangle for the area that we want filtered will not work because the unfiltered video will not be passed outside of the rectangle. The outside area will never change. Also, this program will set the filtered area to be circular which can not be done with processing rectangles.

The idea is to change the constants in the time filter on a pixel by pixel basis as the video stream is being processed. This type of fine control over pipe operations is accomplished with the multiple output state controller (MOSC) elements that are on many devices in the MV200. A MOSC controls the state of an element based on a control signal given to it. On the element flow diagrams the MOSC's are shown as rectangular elements with cross hatched borders and cross hatched control lines exiting from them. The output of a MOSC specifies how many states there are in its controlled element The control signal input to the MOSC is usually a small number of bits, 1 or 2, and the number of controlled states is 2 raised to this number.

The data to control a MOSC comes from data in the MV200. This can be data that was calculated by a previous pipe firing or calculated by the host computer and stored in a surface. The latter technique is the one used in this program whose source code is in mosc.c. The acquisition and display pipes are identical to the ones in average.c. The processing pipe has the additional sections for MOSC control of the time averaging calculation. A flow diagram of the elements related to MOSC control is shown in Figure 7. The idea, as mentioned above, is to control the constants in the time averaging filter on a pixel by pixel basis. The averaging values of K are used in the portion of the image that is to be filtered. In the other parts of the image a K that is equivalent to no time averaging, i.e. K = 1.0, is used.

The operation of the controlled element for each of the MOSC states is set with the *dqSetMoscState()* function to introduce the setting of a state. In this call the MOSC state is specified as a parameter. Setting any attributes, such as constant values or multiplexer selections, for any element controlled by this MOSC will be associated with this MOSC state until the state is changed with another call to *dqSetMoscState()*. There is a MOSC state ALL_STATES that allows the program to specify element attributes that are in effect for all MOSC states. This is equivalent to no MOSC control of the element. Mention of MOSC state in this call is only for ImageFlow program reference. It does not set the MOSC to a certain state. The MOSC state is set by the input signals to the MOSC when the pipe is fired. Based on the MOSC inputs the attributes of the controlled elements will be changed. Again, this control occurs on a pixel by pixel basis in the operating pipe.

In mosc.c a 1 in the FoveaMask surface specifies the region to filter. You can see from the code that AU_L_K1 and AU_L_K0 are set to the time averaging values in MOSC state 1. MOSC state 0 sets constants equivalent to no time averaging. Several ImageFlow graphics functions are used for specifying the mask region. These provide very easy methods to write patterns of pixels to memory surfaces. The program also stops the processing pipe whenever new values are entered for the program parameters. A better method would allow the program to do the operation during the vertical retrace interval. Events and how to use them for synchronizing activities to

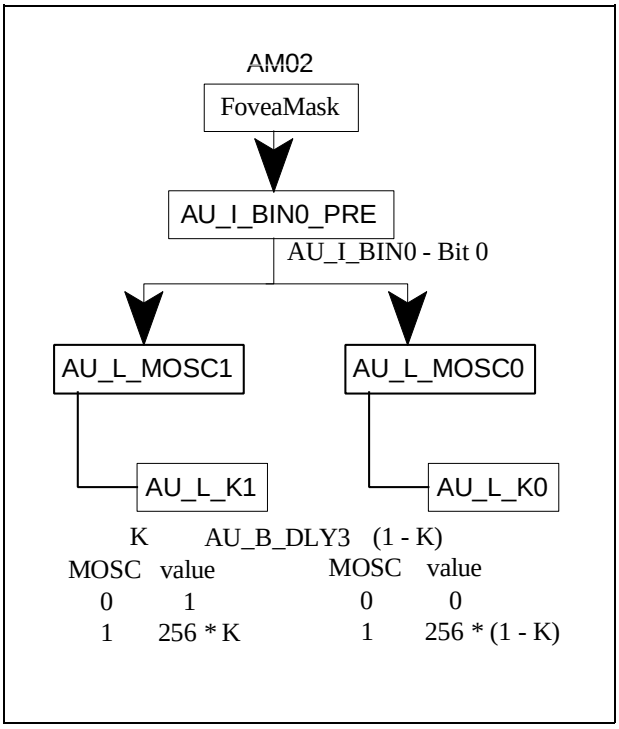


**Figure 7** MOSC control of time filter values in mosc.c

the completion of a pipe firing will be discussed in the next section.

To very clearly see the effects of frame averaging, try setting the averaging constant rather low (< 0.2) and slightly jiggle the camera while on a tripod.

## PAT yourself on the back after this

This section will introduce one of the most important concepts for high speed execution of ImageFlow programs. It is the concept of a pipeline altering thread (PAT). A PAT provides a mechanism to pre-define a pipe topology and parameter setting. The time consuming part of setting up a pipe is the calculation of all the delays and configuration information when the pipe is armed. With a PAT these steps are performed ahead of time and only the compiled results are stored.

ImageFlow statements operate differently when called within a PAT definition. The execution of many statements is deferred until the PAT is executed. The ImageFlow software extracts from each call as much configuration data as possible. This information is stored internally. Execution of a PAT involves the relatively fast process of running the precompiled results. Changes to a pipe's configuration or settings can often be done within the vertical blanking interval of the incoming video signal. In addition, the internals of the MV200 operate at 20 MHz. An RS-170 video signal is a 10 MHz signal. This almost allows the execution of two pipe configurations for each frame of input data.

PAT's can be triggered to run on events. The ImageFlow Event Manager handles detection and setting of these events. An event can be the completion of a pipe firing or the end of a PAT executing either of which the programmer can use to synchronize program flow with the pipeline activity. PAT's can be set to run on an event cyclically or one-shot. You can fire a PAT from within another PAT or specify for the Event Manager to handle this chaining to link several PAT's together into a long sequence of operations. The ultimate in performance for a complex ImageFlow application is gotten when all of the pipe reconfigurations are precalculated in PAT's and are run as a chain by the Event Manager.

The last program is contained in the file multiops.c. It takes the processing done in the previous programs, namely, convolution and recursive frame averaging, and performs them all together in one program. This program requires a high resolution monitor capable of displaying a 1kx1k image. The output is displayed as four images on the monitor for the unmodified, frame averaged, horizontal line convolution and vertical line convolution operations. The program uses PAT's to rapidly reconfigure the hardware between the four processing phases.

The acquisition and display pipes are the same that we have been using for all of the example programs. Two other pipes are also created: one, for the convolutions and a second, for the frame averaging operation. Each of these one-shot pipes will be fired in one of two configurations. The convolution pipe will do the horizontal and vertical line convolutions. The frame averaging pipe will be responsible for displaying the averaged frame and the unmodified image.

The basic idea that you want to follow for fastest operation of an MV200 program is to define all of the non-changing aspects of your pipes first. Then within the PAT execute the commands to make the necessary modifications to the configuration for the operation being set up. The definitions of the four PAT's in this program follow a somewhat standard pattern. The PAT definition starts with a *emBegPat()* call. Following this are the ImageFlow statements that are needed to configure the system for the particular operation to be performed. Any connections or settings that are not modified by running another PAT or from somewhere else in the user program should not be in the scope of the PAT definition. Only things that must be reconfigured when the PAT runs must be specified. For example, the horizontal edge PAT must do the following:

1. attach the horizontal convolution kernel to the NMAC,

2. the output of the AP device must be connected to the display pipe source memory,

3. the alignment point in the destination surface must be adjusted to place the image in the proper place on the display.

ImageFlow statements executed within a PAT definition operate differently than normal. When executed while defining a PAT many statements are run in what is called deferred mode. Internally, ImageFlow compiles the effects of the statement but does not actually modify the hardware to effect the change. The actual change in the configuration will take place only when the PAT is run. Some statements can be executed in a PAT definition but are not deferred, while others are not legal at all. These classes of ImageFlow functions are listed in section 7.2 of the Event Manager chapter in the ImageFlow System manual. The horizontal PAT has both deferred and non-deferred statements executed in it. The statements to perform the steps listed above are in the PAT and are deferred. The *dqCreatePipe()* call to create the convolution pipe is here also. This is not a deferred call and will execute at PAT definition time to create the pipe

object. The creation of the pipe can only be done after a complete pipe has been specified to ImageFlow. Some of that definition is in this PAT, even though the statements are deferred. It would have been possible to put these statements and the *dqCreatePipe()* call with the rest of the definition of the convolution pipe earlier in the program. The PAT would then have to duplicate the calls needed for the reconfiguration. Programmer preference will determine which technique is used.

When the PAT runs it must also arm and fire the pipe after the configuration changes have been made. *dqArmPipe()* is a partially deferred call within a PAT. The time consuming initial part of calculating delays through the pipe is performed but no actual modification to the hardware registers is done until PAT execution. The win with using PAT's is that the loading of the compiled changes is very fast compared to the time for all steps of the arming operation.

*Hint 19*
*Make sure that all connections and settings for the desired operation are in place when the pipe is armed and fired. ImageFlow will not restore to its original state anything that may have been modified by an intervening PAT or ImageFlow statement since the last pipe firing.*

After arming the pipe, a pipe event is gotten that will signal when the pipe has finished a transfer. For a one-shot pipe, a new event must be gotten for each firing of the pipe. ImageFlow arranges that a pipe event gotten within a PAT definition will be valid for each firing of a one-shot pipe during the execution of the precompiled PAT. The final statement in the PAT is a wait for the pipe transfer to complete before ending the PAT. This will ensure that the next changes will not be started until after this operation has finished.

ImageFlow defines two different types of events: normal and reference. The normal event can only be used to detect a cyclic event such as the completion of a transfer on a continuous pipe. Calling *emWaitEvent()* for a one time event is liable to cause ImageFlow to hang if the event has already occurred and was missed. Reference events handle this problem by remembering all events that occur. If a program executes an *emWaitRefEvent()* call and the event has already occurred the call will return immediately. To identify to ImageFlow when you want to begin monitoring an event you call *emMarkRefEvent()*. All event occurrences subsequent to that call will be remembered by ImageFlow and will not be lost to a late *emWaitRefEvent()* call. ImageFlow marks all pipe events as part of the pipe firing operation so they are available as reference events.

After all four PAT's have been defined they are chained together into a loop to cycle one after the other until stopped. The cycle is started by simulating one of the events to get everything going.

## Debugging - you never had it so bad!

Up until now no mention has been made about aids for debugging ImageFlow programs. Without having said anything you have heard about all of it. There basically are no tools for debugging ImageFlow programs! Datacube does provide a few features that occasionally can shed a little light on a problem. These are described in the Tools/Utilities Appendix of the *ImageFlow Reference Manual*. If an error is generated by an ImageFlow function call the particular offending command can usually be tracked by setting the environment variable DQTRACEFUNC to 1. This will turn on a verbose listing of every ImageFlow function call. The last function call listed is the one that generated the error. You do not get any more information about the error but sometimes it is difficult in a large program to even determine just which call generated the error message. The area within your program where function trace information is generated can be controlled with *_dqEnableTraceFunc()* and *_dqDisableTraceFunc()*.

The behind the scenes operations performed by ImageFlow when arming a pipe will be displayed if the environment variable DQTRACEPIPE is set to 1. This output is interesting but even less useful than the function trace. Again, *_dqEnableTracePipe()* and *_dqDisableTracePipe()* can limit the output. PAT's and external cabling have similar utilities described in this section of the manual.

Most of the time you will use the "It looks OK" method of correctness proof. For those more critical applications, test data can be loaded directly into the source surfaces of pipes. The host processor can then fire the pipes and retrieve the output directly from the destination surfaces. This data is available for programmer inspection of the results.

## Conclusions

This report has presented an introduction to the Datacube MV200 and ImageFlow programming. It showed many of the standard techniques used in ImageFlow programs for setting up the MV200 hardware and processing pipes. The example programs used hardware from all the devices but much of the capabilities of the environment remains untouched. Hopefully, with the background learned by reading through this report and working with the example programs, the reader will have the confidence needed to enter the MV200 jungle and do further explorations on their own.

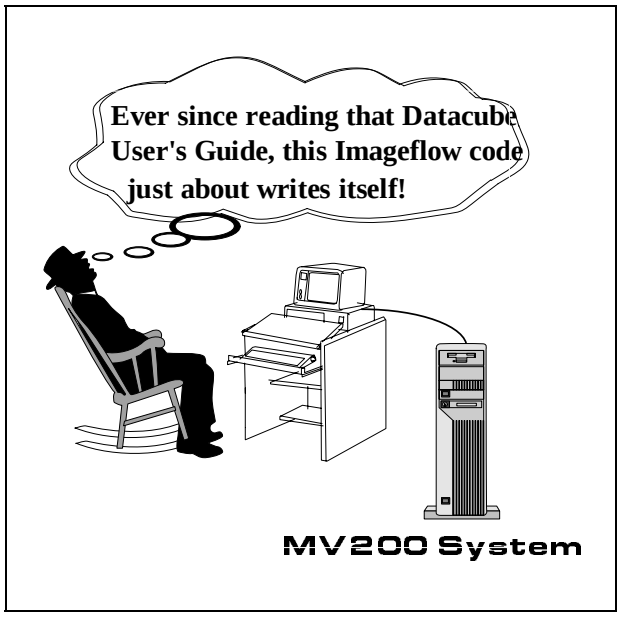


**Figure 8** Typical ImageFlow user after reading this guide

# Glossary of ImageFlow Terms

**Alignment point**. This specifies the pixel position in a memory surface that will be matched with the alignment points of other memory surfaces in the pipe in which this surface exists.

**Arming**. This is the second of three steps that must be performed in order to use a pipe after its topology, i.e. all the connections, have been specified. The function *dqArmPipe()* is used. This step performs most of the calculations needed internally by ImageFlow to use the pipe. A parameter to the call specifies which characteristics of the pipe have changed since the last arming. This can be used to speed up the arming process.

**Configuration File**. The configuration file mainly defines the hardware configuration of the MV200. If you are not using any external hardware the standard configuration file should be sufficient. Otherwise, if you need to use an external board, such as the Digicolor, a file also specifying that device must be used. The configuration file also specifies some ImageFlow software limits, such as, the maximum number of PAT's that can be defined. This will occasionally need to be modified by the programmer for some programs.

**Device**. The major parts of the MV200. Each device is given a two letter designator. All devices used by a program must be identified in the *dqLimitIPDevSet()* declaration and opened with a *dqFindIPDev()* call. The capabilities of a device are described in the device's Functional Description in the *Hardware Reference Manual.*

**Element**. The basic building blocks in the MV200. Most elements correspond directly to pieces of hardware on the MV200. The capabilities of the ImageFlow elements are described in the *Datacube IP Manual*, chapter 5, Datacube Image Processing Elements Model. The elements in a device are shown on that device's flow diagram. The particular capabilities of a device's elements are described in the device's Functional Description and Usage Guide in the *Hardware Reference Manual.*

**Event**. An ImageFlow entity used to identify the occurrence of some activity of interest. Used most often to identify the completion of a pipe or PAT firing. Programs can wait until an event occurs before continuing execution. You can wait for an event with the *emWaitEvent()* function. Events are handled by ImageFlow's em section.

**Firing**. The last thing that must be done to get a pipe running. A pipe is fired using the *dqFirePipe()* function. The pipe will operate in the trigger mode, i.e. continuous or one-shot, that was specified when the pipe was created.

**Gateway**. Every memory device has a receive and transmit gateway. The receive gateway connects from a pipe to a memory and the transmit gateway connects from memory to a pipe. A gateway can be attached to only one memory surface at a time even though multiple memory surfaces may exist in the memory. Receive gateways provide the capability for pixel sub-sampling, while transmit gateways can duplicate pixel data.

**Memory Store**. A memory element stores information in the MV200. There are memory elements throughout the MV200 hardware. Some memory elements are physical RAM memories, such as on the AM devices, while, others are virtual memories, such the A-to-D converter on the AS device. A surface must be created on a memory element to get access to the data in that memory.

**PAT (Pipeline Altering Thread)**. This is a feature of ImageFlow that allows you to predefine alterations to a pipe. The ImageFlow software calculates the necessary changes but defers performing the operations until the PAT is fired. The list of functions that are deferred inside of a PAT definition is given in the *ImageFlow System Manual*, chapter 7, The Event Manager.

**Pipe**. This is the main processing entity in ImageFlow. Multiple pipes can be defined and operate concurrently up to the limits of the timing bus. After the topology of a pipe is specified, the *dqCreatePipe()* function is used to get an id for the pipe. In this call you specify the trigger mode of the pipe as either continuous or one-shot. This can not be changed. Pipes can have multiple sources and/or destinations. All branches of a pipe should start and end at a memory surface.

**Processing Rectangle**. The region of a surface that will be processed by a pipe can be controlled be setting the processing rectangle for that surface. The actual pixels

that are processed is determined by aligning all source surfaces in the pipe with the destination surface and finding the intersection of the processing rectangles. The processing rectangle defaults to the entire surface.

**Ref Event**. When waiting for an event the programmer needs to be concerned about missing a one-shot event, such as, the end of a one-shot pipe transfer. To make sure that ImageFlow keeps track of any non-recurring events that happen before the program begins looking for them, mark the event as a ref event using *emMarkRefEvent()*. After setting it up, wait for the event using *emWaitRefEvent()* rather than *emWaitEvent()*. If the event has already occurred the call to *emWaitRefEvent()* will return immediately.

**System**. This is the object that represents the whole MV200 hardware. The system is defined using either the *dqCreateStdSys()* or *dqCreateSys()* functions. The former creates a standard hardware arrangement based on a standard configuration file. The latter allows you to define other hardware configurations with a specified configuration file.

**Surface**. This is the object that the programmer defines to get access to memory in the MV200. A surface should always begin and end every branch of a pipe. A surface is defined on a memory element which can be standard RAM, an I/O device like the AG's output D-to-A, or the "memory" that holds the kernel for convolutions in the AP's NMAC. For most memory elements, several surfaces can be defined at the same time up to the limits of the size of the element. A memory element, however, can not have more than one surface connected to either its input or output at a time. Attachment is specified with the *dqAttachSurf()* function.

**Timing Bus**. Each pipe defined on the MV200 requires a timing source to keep everything synchronized. The MV200 has four timing busses available. This limits the number of pipes that can be executing concurrently. Normally, ImageFlow handles the timing busses internally. It only becomes an issue for the programmer when more than four pipes are needed. It moves, at that point, into the realm of tricks and ImageFlow folklore to define more pipes.

**Virtual Store.** An element that acts as a source or sink of data and can have a memory surface defined on it. In most respects it looks like a memory store to the program. These are usually associated with devices other than RAM memory such as A/D or D/A convertors.

# Appendix A
# How to Run the MV200 Demo Program

The demo program shows many of the features of the MV200 system. It runs under the Suntools environment and requires a multi-sync monitor capable of displaying 800x600 resolution images.

Use the following steps to run the demo program:

1. Connect a video source to the AS_V0 input of the MV200.

2. Connect a multi-sync monitor to the output of the MV200 that is capable of displaying an 800x600 resolution image.

3. You have to be running in the Suntools environment for the demo program to work:

    a. Exit out of X windows.

    b. Hit Control-C to prevent logging out or do this when you login to prevent the start of your X environment.

    c. Run /usr/bin/suntools.

4. Set the following environment variables:
    a. DQHOME /s8/imageflow
    b. DQDEMO /s8/imageflow/demo

5. cd to /s8/imageflow/bin and run mv200demo.

6. The mouse is active on the MV200 output display. To get back to Suntools push the Suntools button. To exit the demo press the Quit button.

7. There are several levels of menus that can be explored. Some settings, most notably the filtering, will be preserved across menu choices. Experiment with the different operations demonstrated by the program. Be forewarned, that there are one or two bugs that can cause the program to abort.

# Appendix B
# MV200 Element Flow Diagrams

**These element flow diagrams from the MaxVideo 200 Hardware Reference manual are included with the permission of Datacube, Inc.**

Please refer to the following Datacube MV200 Element Flow Diagrams:

1. AB Device Element Flow Diagram — AB 2-7/2-8
2. AG Device Element Flow Diagram — AG 2-7/2-8
3. AM Device Element Flow Diagram — AM 2-7/2-8
4. AP Device Element Flow Diagram (Showing Default Configuration Blocks) — AP 2-7/2-8
5. AS Device Element Flow Diagram — AS 2-7
6. AU Device Element Flow Diagram (Input Section) — AU 2-9/2-10
7. AU Device Element Flow Diagram (Binary Crosspoint, Grey Scale Crosspoint and Output Sections) — AU 2-11/2-12
8. AU Device Element Flow Diagram (Linear Processor Section) — AU 2-13/2-14

# Appendix C
# Example Program Listings

## simple.c

```
/************************************************************************

    simple.c - pass data through the MV200

    This is a simple first ImageFlow program.  It passes a video signal
    through the MV200 by digitizing it, storing the data in a memory,
    reading it back out, and converting it back to analog video.

************************************************************************/

#include <stdio.h>
#include <datacube.h>                   /* this file is always included */

/* define the display size */
#define DISPLAYXSIZE    512L
#define DISPLAYYSIZE    484L

/* you have to list all devices that the program will use */
/* DC is here only because it is in the chasis as SPC master */
dqLimitIPDevSet(AB AM AS AG DC);

main()
{
    DqSystem oSystem;
    DqIPDev oAb00, oAm00, oAg00, oAs00;
    DqSurf oDispSrcSurf, oDispDstSurf, oAcqSrcSurf, oAcqDstSurf;
    DqPipe oDispPipe, oAcqPipe;

    char pcUserInpBuf[80];

    /* initialize the hardware and ImageFlow software */
    dqInitEnv();

    /* get handles for the system and all devices being used */
    oSystem = dqCreateStdSys();
    oAb00 = dqFindIPDev(oSystem, "ab00");
    oAs00 = dqFindIPDev(oSystem, "as00");
    oAg00 = dqFindIPDev(oSystem, "ag00");

    /* memory is referenced a little differently. It is considered part of
     the AB device. */
    oAm00 = dqFindIPDev(oSystem, "ab00:am00");


    /*********************  ACQUISITION PIPE ****************************

        Define the source surface, attach a gateway to it, route it across
        the cross point switch, attach it to a destination memory and
        surface. */

    /* 512x484 is the std size surface on the AS device */
    oAcqSrcSurf = dqCreateStdSizeSurf(oAs00, AS_ADC);
    dqAttachSurf(oAcqSrcSurf, AS_XMT);

    dqConnect(oAb00, DQ_CSG, AB_OP00);      /* use unsigned output of AS */

    /* memories have some internal hardware that must be set up correctly
        even just to get something in or out of them */
    amSetRcvGateway20MHz(oAm00);

  /* this will be the end of the acquisition pipe */
    oAcqDstSurf = dqCreateSameSizeSurf(oAm00, AM_MEM_R8,oAcqSrcSurf);
    dqAttachSurf(oAcqDstSurf,AM_RCV);

    /* create a continuously running pipe, arm and fire it */
    oAcqPipe = dqCreatePipe(oAcqDstSurf, DQ_TRG_CONTINUOUS);
    dqArmPipe(oAcqPipe, DQ_DSM_PIPE);
    dqFirePipe(oAcqPipe);


    /*********************** DISPLAY PIPE ******************************

       Define the display source surface, attach a gateway to it, route it
       across the cross point switch, attach it the display memory and get
       it running. */

    /* dup'ing the surface makes this new surface be the same physical
        memory as the other surface.  It is not allocated from a new
        section of memory */
    oDispSrcSurf = dqDupSurf(oAcqDstSurf);
    dqAttachSurf(oDispSrcSurf,AM_XMT);

    /* the gateways into the AG memory always are set to run 40MHz.
        If you are sending data from a memory back to the cross point
        switch then the gateway is set for 20MHz. */
    amSetDispGateway40MHz(oAm00);

    /* because the AG gateway always runs at 40MHz you have to make sure
        that you provide a flow of 40MHz data.  The AG manual tell you
        what factors to use based on the type of display that you are
        using. */
    dqSpecXmtExpansion(oAm00,AM_XMT,4,1);


  /* connections within the AG device */
    dqConnect(oAg00, AG_RED, AG_DAC_LUT_SRC);
    dqConnect(oAg00, AG_DAC_LUT, AG_DAC_SRC);
    oDispDstSurf = dqCreateSurf(oAg00, AG_DAC,DISPLAYXSIZE,DISPLAYYSIZE);
    dqAttachSurfGate(oDispDstSurf, AG_RCV);

    /* take away the dummy expansion */
    dqSpecRcvShrinkage(oAg00,AG_RCV,4,1);

    /* create the display pipe, arm and fire it */
    oDispPipe = dqCreatePipe(oDispDstSurf, DQ_TRG_CONTINUOUS);
    dqArmPipe(oDispPipe, DQ_DSM_PIPE);
    dqFirePipe(oDispPipe);

    printf("Hit any key to exit.\n");
    fgets(pcUserInpBuf, 80, stdin);

    dqDisposeSys(oSystem);
}



/* auxiliary functions to set up the AM gateways to the correct speed */
amSetRcvGateway20MHz(oAmDev)
DqIPDev oAmDev;
{
    dqConnect(oAmDev,AM_SPUP0,AM_INPUT0);
    dqConnect(oAmDev,AM_INPUT0,AM_OP0);
    dqSpecLogic(oAmDev,AM_LOGIC4,0,1);
    dqSpecLogic(oAmDev,AM_LOGIC0,0xffffffff,0);
    amSetGateSysClkMult(oAmDev,AM_RCV,2);
}


amSetDispGateway40MHz(oAmDev)
DqIPDev oAmDev;
```

**simple.c**

```
{
    amSetGateSysClkMult(oAmDev,AM_XMT,4);
    dqConnect(oAmDev,AM_XMT_OUT,AM_OP3);
    dqSpecLogic(oAmDev,AM_LOGIC3,0xff,0);
    dqConnect(oAmDev,AM_LOGIC3,AM_OUTPUT0);
    dqConnect(oAmDev,AM_XMT,AM_XMT_OUT);
}
```

## convolve.c

```
/************************************************************************

    convolve.c - perform a 3x3 convolution on a video data stream

    This program will define three pipes on the MV200: an acquisition,
    processing, and display pipe.  The processing pipe will use the NMAC
    element on the AP device to perform a video rate 3x3 convolution.

************************************************************************/

#include <stdio.h>
#include <datacube.h>

#define DISPLAYXSIZE    512L
#define DISPLAYYSIZE    484L

/* convolution pattern */
#define KERNELXSIZE 3
#define KERNELYSIZE 3


/* identify the devices used in this program */
/* DC is here only because it is in the chasis as SPC master */
dqLimitIPDevSet(AB AM AS AG AP DC);


main()
{
    DqSystem oSystem;
    DqIPDev oAb00, oAg00, oAs00;
    DqIPDev oAm00, oAm01, oAp00;
    DqSurf oAcqSrcSurf, oAcqDstSurf;
    DqSurf oProcSrcSurf, oProcDstSurf, oKernel;
    DqSurf oDispSrcSurf, oDispDstSurf;
    DqPipe oDispPipe, oAcqPipe, oProcPipe;
    DqRect tConvRect;               /* convolution rectangle description */

  DqByte Coefs[3][3];                   /* 3x3 convolution kernel coefficients */
    char pcUserInpBuf[80];

    /* initialize the system and get handles for the devices */
    dqInitEnv();

    oSystem = dqCreateStdSys();

    oAb00 = dqFindIPDev(oSystem, "ab00");
    oAs00 = dqFindIPDev(oSystem, "as00");
    oAg00 = dqFindIPDev(oSystem, "ag00");
    oAp00 = dqFindIPDev(oSystem, "ap00");

    oAm00 = dqFindIPDev(oSystem, "ab00:am00");
    oAm01 = dqFindIPDev(oSystem, "ab00:am01");

  /* set the data type checking to tolerant */
    dqSetDTPDogma(DQ_DTP_TOLERANT);

    /************************ ACQUISITION PIPE *************************/
    oAcqSrcSurf = dqCreateStdSizeSurf(oAs00, AS_ADC);
    dqAttachSurf(oAcqSrcSurf, AS_XMT);

    /* take signed data from the AS device */
    dqSetDTT(oAs00, AS_DTM, DQ_DTT_X_SIGNED);

    /* use DQ_CSR for signed output from the AS device */
    dqConnect(oAb00, DQ_CSR, AB_OP00);

    amSetRcvGateway20MHz(oAm00);
    oAcqDstSurf = dqCreateSameSizeSurf(oAm00, AM_MEM_R8,oAcqSrcSurf);
    dqAttachSurf(oAcqDstSurf,AM_RCV);

    /* create a continuously running pipe, arm and fire it off */
    oAcqPipe = dqCreatePipe(oAcqDstSurf, DQ_TRG_CONTINUOUS);
    dqArmPipe(oAcqPipe, DQ_DSM_PIPE);
    dqFirePipe(oAcqPipe);


    /********************** PROCESSING PIPE ***************************/

    /* start with the data from the acquisition pipe */
    oProcSrcSurf = dqDupSurf(oAcqDstSurf);
    dqAttachSurf(oProcSrcSurf, AM_XMT);

    /* because this is running into the cross point for processing the
        memory must output at 20 MHz. */
    amSetXmtGateway20MHz(oAm00);

    dqConnect(oAb00, DQ_IMX0, AB_OP21);     /* across to the NMAC input */

    /* set the configuration for the NMAC element to 8x8 convolution */
    dqConnect(oAp00, AP_SHIFT8, AP_NDLY_SRC);

    /* create a surface to define the kernel in the NMAC */
    oKernel = dqCreateSurf(oAp00, AP_NMAC8, KERNELXSIZE, KERNELYSIZE);
    dqSetSurfBaseDT(oKernel, DQ_DT_SIGNED); /* everything will be signed */
    dqAttachSurf(oKernel, AP_NMAC8);    /* you have to attach to this also */

    /* the host will write directly into the surface with the coeffient
        values.  This rectangle is defined with respect to the center point.
        The default center point is the center pixel of the kernel. */
    tConvRect.lXMin = -(KERNELXSIZE / 2);
    tConvRect.lXMax = KERNELXSIZE % 2 ? KERNELXSIZE / 2
                                      : KERNELXSIZE / 2 - 1;
    tConvRect.lYMin = -(KERNELYSIZE / 2);
    tConvRect.lYMax = KERNELYSIZE % 2 ? KERNELYSIZE / 2
                                      : KERNELYSIZE / 2 - 1;

    /* this defines a 3x3 horizontal line filter */
    Coefs[0][0] = 64;
    Coefs[0][1] = 64;
    Coefs[0][2] = 64;
    Coefs[1][0] = 0;
    Coefs[1][1] = 0;
    Coefs[1][2] = 0;
    Coefs[2][0] = -64;
    Coefs[2][1] = -64;
    Coefs[2][2] = -64;

    dqWtRect(oKernel, &tConvRect, Coefs);

    /* define constants that feed the AP_ADD B operand */
    dqSetKVal(oAb00,AB_OP17,0L);
    dqSetKVal(oAb00,AB_OP18,0L);
    dqSetKVal(oAb00,AB_OP19,0L);
  dqSetOpndBaseDT(oAp00,AP_ADD,DQ_OPND_A,DQ_DT_SIGNED);

    /* shift so that the 8-bit result comes out at DQ_CP15.  This
        shifting must also be accounted for when setting the NMAC
        coefficients */
    dqSpecShift(oAp00, AP_SHIFT8, DQ_SHIFT_ARITHMETIC, 17);
    dqSpecShift(oAp00, AP_SHIFT,  DQ_SHIFT_ARITHMETIC,  7);
```

## convolve.c

```
  /* now set the output format to account for the convolution.
     Horizontal edges are convolved toward -127 and 127. Use the
     absolute value output so that both edges are driven toward
     127.  This also shifts the values up one place. */
  dqSetOpndBaseDT(oAp00,AP_FORMAT,DQ_OPND_A,DQ_DT_SIGNED);
  dqSetFormatOp(oAp00,AP_FORMAT,AP_FMT_ABSOLUTE);

    /* across the cross point switch into AM01 */
    dqConnect(oAb00, DQ_CP15, AB_OP01);
    amSetRcvGateway20MHz(oAm01);

    oProcDstSurf = dqCreateSameSizeSurf(oAm01, AM_MEM_R8,oProcSrcSurf);
    dqAttachSurf(oProcDstSurf,AM_RCV);

    /* create a continuously running pipe, arm and fire it off */
    oProcPipe = dqCreatePipe(oProcDstSurf, DQ_TRG_CONTINUOUS);
    dqArmPipe(oProcPipe, DQ_DSM_PIPE);
    dqFirePipe(oProcPipe);



    /********************** DISPLAY PIPE ******************************/
    oDispSrcSurf = dqDupSurf(oProcDstSurf);
    dqAttachSurf(oDispSrcSurf,AM_XMT);

    amSetDispGateway40MHz(oAm01);
    dqSpecXmtExpansion(oAm01,AM_XMT,4,1);

    dqConnect(oAg00, AG_GREEN, AG_DAC_LUT_SRC);
    dqConnect(oAg00, AG_DAC_LUT, AG_DAC_SRC);

    oDispDstSurf = dqCreateSurf(oAg00, AG_DAC,DISPLAYXSIZE,DISPLAYYSIZE);
    dqAttachSurfGate(oDispDstSurf, AG_RCV);
    dqSpecRcvShrinkage(oAg00,AG_RCV,4,1);

    oDispPipe = dqCreatePipe(oDispDstSurf, DQ_TRG_CONTINUOUS);
    dqArmPipe(oDispPipe, DQ_DSM_PIPE);
    dqFirePipe(oDispPipe);


    printf("Hit any key to exit.\n");
    fgets(pcUserInpBuf, 80, stdin);

    dqDisposeSys(oSystem);
}



/* auxiliary functions to set up the AM gateways to the correct speed */
amSetXmtGateway20MHz(oAmDev)
DqIPDev oAmDev;
{
    amSetGateSysClkMult(oAmDev,AM_XMT,2);
    dqConnect(oAmDev,AM_XMT_OUT,AM_OP3);
    dqSpecLogic(oAmDev,AM_LOGIC3,0xffffffff,0);
    dqConnect(oAmDev,AM_SLDN0_T0,AM_OUTPUT0);
    dqConnect(oAmDev,AM_XMT,AM_XMT_OUT);
}


amSetRcvGateway20MHz(oAmDev)
DqIPDev oAmDev;
{
    dqConnect(oAmDev,AM_SPUP0,AM_INPUT0);
    dqConnect(oAmDev,AM_INPUT0,AM_OP0);
    dqSpecLogic(oAmDev,AM_LOGIC4,0,1);
    dqSpecLogic(oAmDev,AM_LOGIC0,0xffffffff,0);
    amSetGateSysClkMult(oAmDev,AM_RCV,2);
}


amSetDispGateway40MHz(oAmDev)
DqIPDev oAmDev;
{
    amSetGateSysClkMult(oAmDev,AM_XMT,4);
    dqConnect(oAmDev,AM_XMT_OUT,AM_OP3);
    dqSpecLogic(oAmDev,AM_LOGIC3,0xff,0);
    dqConnect(oAmDev,AM_LOGIC3,AM_OUTPUT0);
    dqConnect(oAmDev,AM_XMT,AM_XMT_OUT);
}
```

## procrect.c

```
/************************************************************************
    procrect.c - perform a 3x3 convolvution in a processing rectangle

    This program will define three pipes on the MV200: an acquisition,
    processing, and display pipe.  The processing pipe will use the NMAC
    element on the AP device to perform a video rate 3x3 convolution. A
    processing rectangle and alignment point are specified for the
    destination surface in the processing pipe.

************************************************************************/

#include <stdio.h>
#include <datacube.h>

#define DISPLAYXSIZE    512L
#define DISPLAYYSIZE    484L

/* convolution pattern */
#define KERNELXSIZE 3
#define KERNELYSIZE 3


/* identify the devices used in this program */
/* DC is here only because it is in the chasis as SPC master */
dqLimitIPDevSet(AB AM AS AG AP DC);


main()
{
    DqSystem oSystem;
    DqIPDev oAb00, oAg00, oAs00;
    DqIPDev oAm00, oAm01, oAp00;
    DqSurf oAcqSrcSurf, oAcqDstSurf;
    DqSurf oProcSrcSurf, oProcDstSurf, oKernel;
    DqSurf oDispSrcSurf, oDispDstSurf;
    DqPipe oDispPipe, oAcqPipe, oProcPipe;
    DqRect tConvRect;               /* convolution rectangle description */
    DqRect ProcRect;

  DqByte Coefs[3][3];                   /* 3x3 convolution kernel coefficients */
    char pcUserInpBuf[80];

  int RectSize,XPos,YPos;      /* alignment and processing rectangle info */

    /* initialize the system and get handles for the devices */
    dqInitEnv();

    oSystem = dqCreateStdSys();

    oAb00 = dqFindIPDev(oSystem, "ab00");
    oAs00 = dqFindIPDev(oSystem, "as00");
    oAg00 = dqFindIPDev(oSystem, "ag00");
    oAp00 = dqFindIPDev(oSystem, "ap00");

    oAm00 = dqFindIPDev(oSystem, "ab00:am00");
    oAm01 = dqFindIPDev(oSystem, "ab00:am01");


  /* set the data type checking to tolerant */
    dqSetDTPDogma(DQ_DTP_TOLERANT);

    /*********************** ACQUISITION PIPE ************************/
    oAcqSrcSurf = dqCreateStdSizeSurf(oAs00, AS_ADC);
    dqAttachSurf(oAcqSrcSurf, AS_XMT);

    /* take signed data from the AS device */
    dqSetDTT(oAs00, AS_DTM, DQ_DTT_X_SIGNED);

    /* use DQ_CSR for signed output from the AS device */
    dqConnect(oAb00, DQ_CSR, AB_OP00);
    amSetRcvGateway20MHz(oAm00);
    oAcqDstSurf = dqCreateSameSizeSurf(oAm00, AM_MEM_R8,oAcqSrcSurf);
    dqAttachSurf(oAcqDstSurf,AM_RCV);

    /* create a continuously running pipe, arm and fire it off */
    oAcqPipe = dqCreatePipe(oAcqDstSurf, DQ_TRG_CONTINUOUS);
    dqArmPipe(oAcqPipe, DQ_DSM_PIPE);
    dqFirePipe(oAcqPipe);


    /********************** PROCESSING PIPE ****************************/

    /* start with the data from the acquisition pipe */
    oProcSrcSurf = dqDupSurf(oAcqDstSurf);
    dqAttachSurf(oProcSrcSurf, AM_XMT);

    /* because this is running into the cross point for processing the
        memory must output at 20 MHz. */
    amSetXmtGateway20MHz(oAm00);

    dqConnect(oAb00, DQ_IMX0, AB_OP21);     /* across to the NMAC input */

    /* set the configuration for the NMAC element to 8x8 convolution */
    dqConnect(oAp00, AP_SHIFT8, AP_NDLY_SRC);

    /* create a surface to define the kernel in the NMAC */
    oKernel = dqCreateSurf(oAp00, AP_NMAC8, KERNELXSIZE, KERNELYSIZE);
    dqSetSurfBaseDT(oKernel, DQ_DT_SIGNED); /* everything will be signed */
    dqAttachSurf(oKernel, AP_NMAC8);    /* you have to attach to this also */

    /* the host will write directly into the surface with the coeffient
        values.  This rectangle is defined with respect to center point.
        The default center point is the center pixel of the kernel. */
    tConvRect.lXMin = -(KERNELXSIZE / 2);
    tConvRect.lXMax = KERNELXSIZE % 2 ? KERNELXSIZE / 2
                                      : KERNELXSIZE / 2 - 1;
    tConvRect.lYMin = -(KERNELYSIZE / 2);
    tConvRect.lYMax = KERNELYSIZE % 2 ? KERNELYSIZE / 2
                                      : KERNELYSIZE / 2 - 1;

    /* this defines a 3x3 horizontal line filter */
    Coefs[0][0] = 64;
    Coefs[0][1] = 64;
    Coefs[0][2] = 64;
    Coefs[1][0] = 0;
    Coefs[1][1] = 0;
    Coefs[1][2] = 0;
    Coefs[2][0] = -64;
    Coefs[2][1] = -64;
    Coefs[2][2] = -64;

    dqWtRect(oKernel, &tConvRect, Coefs);


    /* define constants that feed the AP_ADD B operand */
    dqSetKVal(oAb00,AB_OP17,0L);
    dqSetKVal(oAb00,AB_OP18,0L);
    dqSetKVal(oAb00,AB_OP19,0L);
  dqSetOpndBaseDT(oAp00,AP_ADD,DQ_OPND_A,DQ_DT_SIGNED);
```

```
  /* shift so that the 8-bit result comes out at DQ_CP15.  This
      shifting must also be accounted for when setting the NMAC
      coefficients */
  dqSpecShift(oAp00, AP_SHIFT8, DQ_SHIFT_ARITHMETIC, 17);
  dqSpecShift(oAp00, AP_SHIFT,  DQ_SHIFT_ARITHMETIC,  7);

/* now set the output format to account for the convolution.
   Horizontal edges are convolved toward -127 and 127. Use the
   absolute value output so that both edges are driven toward
   127.  This also shifts the values up one place. */
dqSetOpndBaseDT(oAp00,AP_FORMAT,DQ_OPND_A,DQ_DT_SIGNED);
dqSetFormatOp(oAp00,AP_FORMAT,AP_FMT_ABSOLUTE);

  /* across the cross point switch into AM01 */
  dqConnect(oAb00, DQ_CP15, AB_OP01);
  amSetRcvGateway20MHz(oAm01);

  oProcDstSurf = dqCreateSameSizeSurf(oAm01, AM_MEM_R8,oProcSrcSurf);
  dqAttachSurf(oProcDstSurf,AM_RCV);

  /* create a continuously running pipe, arm and fire it off */
  oProcPipe = dqCreatePipe(oProcDstSurf, DQ_TRG_CONTINUOUS);
  dqArmPipe(oProcPipe, DQ_DSM_PIPE);
  dqFirePipe(oProcPipe);


  /********************** DISPLAY PIPE ******************************/
  oDispSrcSurf = dqDupSurf(oProcDstSurf);
  dqAttachSurf(oDispSrcSurf,AM_XMT);

  amSetDispGateway40MHz(oAm01);
  dqSpecXmtExpansion(oAm01,AM_XMT,4,1);

  dqConnect(oAg00, AG_GREEN, AG_DAC_LUT_SRC);
  dqConnect(oAg00, AG_DAC_LUT, AG_DAC_SRC);

  oDispDstSurf = dqCreateSurf(oAg00, AG_DAC,DISPLAYXSIZE,DISPLAYYSIZE);
  dqAttachSurfGate(oDispDstSurf, AG_RCV);
  dqSpecRcvShrinkage(oAg00,AG_RCV,4,1);

  oDispPipe = dqCreatePipe(oDispDstSurf, DQ_TRG_CONTINUOUS);
  dqArmPipe(oDispPipe, DQ_DSM_PIPE);
  dqFirePipe(oDispPipe);



  /*********************************************************************/
  /*                                                                   */
  /*  BEGIN ADDED CODE FOR THE PROCESSING RECTANGLE PROGRAM            */
  /*                                                                   */
  /*********************************************************************/

printf("Specify the size of the processing rectangle: ");
scanf("%d",&RectSize);

printf("Specify x position for alignment: ");
scanf("%d",&XPos);

printf("Specify y position for alignment: ");
scanf("%d",&YPos);

/* stop the processing */
dqHaltPipe(oProcPipe);

  /* set the alignment point and processing rectangle */
    dqSpecSurfAlignPoint(oProcDstSurf,XPos,YPos);
    ProcRect.lXMin = ProcRect.lYMin = 0;
    ProcRect.lXMax = ProcRect.lYMax = RectSize;
    dqSetSurfProcRect(oProcDstSurf,&ProcRect);

  /* start the processing pipe again */
  dqArmPipe(oProcPipe,DQ_DSM_RECT);
  dqFirePipe(oProcPipe);

    /*********************************************************************/
    /*                                                                   */
    /*  END ADDED CODE FOR THE PROCESSING RECTANGLE PROGRAM              */
    /*                                                                   */
    /*********************************************************************/

  gets(pcUserInpBuf);                                   /* clear left over '\n' */
  printf("hit any key to exit.\n");
  fgets(pcUserInpBuf,80,stdin);

    dqDisposeSys(oSystem);
}



/* auxiliary functions to set up the AM gateways to the correct speed */
amSetXmtGateway20MHz(oAmDev)
DqIPDev oAmDev;
{
    amSetGateSysClkMult(oAmDev,AM_XMT,2);
    dqConnect(oAmDev,AM_XMT_OUT,AM_OP3);
    dqSpecLogic(oAmDev,AM_LOGIC3,0xffffffff,0);
    dqConnect(oAmDev,AM_SLDN0_T0,AM_OUTPUT0);
    dqConnect(oAmDev,AM_XMT,AM_XMT_OUT);
}


amSetRcvGateway20MHz(oAmDev)
DqIPDev oAmDev;
{
    dqConnect(oAmDev,AM_SPUP0,AM_INPUT0);
    dqConnect(oAmDev,AM_INPUT0,AM_OP0);
    dqSpecLogic(oAmDev,AM_LOGIC4,0,1);
    dqSpecLogic(oAmDev,AM_LOGIC0,0xffffffff,0);
    amSetGateSysClkMult(oAmDev,AM_RCV,2);
}


amSetDispGateway40MHz(oAmDev)
DqIPDev oAmDev;
{
    amSetGateSysClkMult(oAmDev,AM_XMT,4);
    dqConnect(oAmDev,AM_XMT_OUT,AM_OP3);
    dqSpecLogic(oAmDev,AM_LOGIC3,0xff,0);
    dqConnect(oAmDev,AM_LOGIC3,AM_OUTPUT0);
    dqConnect(oAmDev,AM_XMT,AM_XMT_OUT);
}
```

## average.c

```
/*************************************************************************

    average.c - time average a video stream

    This program will apply a recursive filter to time average the video
    data.  The filter that will be applied is:

      y(n) = K*x(n) + (1 - K)*y(n - 1)

      where K is a constant between 0 and 1
            y(n) is the current averaged frame
            y(n-1) is the previous averaged frame
            x(n) is the current input frame

*************************************************************************/

#include <stdio.h>
#include <datacube.h>

/* define the display size */
#define DISPLAYXSIZE    512L
#define DISPLAYYSIZE    484L

/* DC is here only because it is in the chasis as SPC master */
dqLimitIPDevSet(AB AM AS AG AU DC);



main()
{
    DqSystem oSystem;
    DqIPDev oAb00, oAm00, oAm01, oAm05, oAg00, oAs00, oAu00;
    DqSurf oDispSrcSurf, oDispDstSurf, oAcqSrcSurf, oAcqDstSurf;
    DqSurf oProcSrcSurf,oProcDstSurfs[3],LastFrameSrc;
    DqPipe oDispPipe, oAcqPipe, oProcPipe;

    /* initialize the hardware and ImageFlow software */
    dqInitEnv();

    /* get handles for the system and all devices being used */
    oSystem = dqCreateStdSys();
    oAb00 = dqFindIPDev(oSystem, "ab00");
    oAs00 = dqFindIPDev(oSystem, "as00");
    oAg00 = dqFindIPDev(oSystem, "ag00");
    oAu00 = dqFindIPDev(oSystem, "au00");

    oAm00 = dqFindIPDev(oSystem, "ab00:am00");
    oAm01 = dqFindIPDev(oSystem, "ab00:am01");
    oAm05 = dqFindIPDev(oSystem, "ab00:am05");

  /* set the data type checking to tolerant */
    dqSetDTPDogma(DQ_DTP_TOLERANT);


    /********************  ACQUISITION PIPE ****************************/
    oAcqSrcSurf = dqCreateStdSizeSurf(oAs00, AS_ADC);
    dqAttachSurf(oAcqSrcSurf, AS_XMT);

    dqConnect(oAb00, DQ_CSG, AB_OP05);      /* use unsigned output of AS */

    amSetRcvGateway20MHz(oAm05);

    oAcqDstSurf = dqCreateSameSizeSurf(oAm05, AM_MEM_R8,oAcqSrcSurf);
    dqAttachSurf(oAcqDstSurf,AM_RCV);

/* create a continuously running pipe, arm and fire it */
oAcqPipe = dqCreatePipe(oAcqDstSurf, DQ_TRG_CONTINUOUS);
dqArmPipe(oAcqPipe, DQ_DSM_PIPE);
dqFirePipe(oAcqPipe);


/********************** PROCESSING PIPE ****************************/
/* start the video side of the process pipe with a dup of the
   acquisition pipe destination */
oProcSrcSurf = dqDupSurf(oAcqDstSurf);
dqAttachSurf(oProcSrcSurf, AM_XMT);
amSetXmtGateway20MHz(oAm05);

/* data throughout the program will be unsigned */
dqSetSurfBaseDT(oProcSrcSurf,DQ_DT_UNSIGNED);

/* across the cross point to the time filter */
dqConnect(oAb00,DQ_IMX5,AB_OP08);        /* current frame into Moe */
dqSetKVal(oAb00,AB_OP09,0);
dqConnect(oAu00,AU_I_SEP_MOE,AU_I_MOE);
dqConnect(oAu00,AU_I_MOE_X0,AU_G_OP8);

/* create K * x(n)  K set in AU_L_K1 */
dqConnect(oAu00,AU_L_EXT0,AU_L_MULT1_BOP);
dqSetKBaseDT(oAu00,AU_L_K1,DQ_DT_SIGNED);
dqConnect(oAu00,AU_L_K1,AU_L_MULT1_AOP);

/* create surface to save the last frame */
LastFrameSrc = dqCreateSameSizeSurf(oAm01,AM_MEM_R8,oProcSrcSurf);
dqSetSurfBaseDT(LastFrameSrc,DQ_DT_UNSIGNED);
dqAttachSurf(LastFrameSrc,AM_XMT);
amSetXmtGateway20MHz(oAm01);

/* bring last frame across the cross point to the time filter */
dqConnect(oAb00,DQ_IMX1,AB_OP10);        /* last frame into Larry */
dqSetKVal(oAb00,AB_OP11,0);
dqConnect(oAu00,AU_I_SEP_LARRY,AU_I_LARRY);
dqConnect(oAu00,AU_I_LARRY_X0,AU_G_OP9);

/* create (1 - K) * y(n - 1)  (1 - K) set in AU_L_K0 */
dqConnect(oAu00,AU_L_EXT1,AU_L_MULT0_AOP);
dqSetKBaseDT(oAu00,AU_L_K0,DQ_DT_SIGNED);
dqConnect(oAu00,AU_L_K0,AU_L_MULT0_BOP);

/* zero down the AU_L_MULT2 and AU_L_MULT3 chains */
dqSetKVal(oAu00,AU_L_K2,0x00);
dqConnect(oAu00,AU_L_K2,AU_L_MULT2_AOP);
dqConnect(oAu00,AU_L_K2,AU_L_MULT2_BOP);
dqSetKVal(oAu00,AU_L_K3,0x00);
dqConnect(oAu00,AU_L_K3,AU_L_MULT3_AOP);
dqConnect(oAu00,AU_L_K3,AU_L_MULT3_BOP);

/*
  Combine to get K*x(n) + (1 - K)*y(n - 1)
  Calculations are performed with binary point at bit 7-8
  This shift moves binary point to bit 0-1 when leaving AU_L_SHIFT3
*/
dqSpecShift(oAu00,AU_L_SHIFT3,DQ_SHIFT_ARITHMETIC,-7);

/* Then effect rounding by adding 1 before AU_L_ADD3 tosses the LSB */
dqSetKVal(oAu00,AU_L_K4,1);
dqConnect(oAu00,AU_L_K4,AU_L_ADD3_BOP);

/* now straight shot to AU_L_RESULT */
dqConnect(oAu00,AU_L_CLIP,AU_L_P_RES);
```

```
    dqConnect(oAu00,AU_L_P_RES,AU_L_RESULT);

    /* pass result back through the AU cross point */
    dqConnect(oAu00,AU_L_RESULT_X0,AU_G_OP1);
    dqConnect(oAu00,AU_O_CLIP1,AU_O_C7_SRC);

    /* across cross point to display and last frame surface */
    dqConnect(oAb00,DQ_CU07,AB_OP01);
    dqConnect(oAb00,DQ_CU07,AB_OP00);

    /* create surface to save this frame for next time */
    oProcDstSurfs[0] = dqDupSurf(LastFrameSrc);
    dqAttachSurf(oProcDstSurfs[0],AM_RCV);
    amSetRcvGateway20MHz(oAm01);

    /* create surface for the display output */
    oProcDstSurfs[1] = dqCreateSameSizeSurf(oAm00,AM_MEM_R8,oProcSrcSurf);
    dqAttachSurf(oProcDstSurfs[1],AM_RCV);
    amSetRcvGateway20MHz(oAm00);

    /* terminate surface list and create multi-destination pipe */
    oProcDstSurfs[2] = 0;
    oProcPipe = dqCreateMultiDstPipe(oProcDstSurfs,DQ_TRG_CONTINUOUS);
    dqArmPipe(oProcPipe,DQ_DSM_PIPE);
    dqFirePipe(oProcPipe);


    /********************** DISPLAY PIPE ******************************/

    oDispSrcSurf = dqDupSurf(oProcDstSurfs[1]);
    dqAttachSurf(oDispSrcSurf,AM_XMT);
    dqSpecXmtExpansion(oAm00,AM_XMT,4,1);

    amSetDispGateway40MHz(oAm00);

    dqConnect(oAg00, AG_RED, AG_DAC_LUT_SRC);
    dqConnect(oAg00, AG_DAC_LUT, AG_DAC_SRC);
    oDispDstSurf = dqCreateSurf(oAg00, AG_DAC,DISPLAYXSIZE,DISPLAYYSIZE);
    dqAttachSurfGate(oDispDstSurf, AG_RCV);

    dqSpecRcvShrinkage(oAg00,AG_RCV,4,1);

    /* create the display pipe, arm and fire it */
    oDispPipe = dqCreatePipe(oDispDstSurf, DQ_TRG_CONTINUOUS);
    dqArmPipe(oDispPipe, DQ_DSM_PIPE);
    dqFirePipe(oDispPipe);


/* get the values for the time filter average. */
    do
    {
        float NewK;
        int K,OneMinusK;

        printf("Enter K between 0.0 and 1.0 (<0 to stop): ");
        scanf("%f",&NewK);
        if(NewK < 0.0)
            break;
        if(NewK > 1.0)
            continue;

     /* these constants take into account that the calculations are
        performed with the binary point between bits 7-8 */
        K = irint(256.0 * NewK);
        OneMinusK = irint(256.0 * (1.0 - NewK));

        /* load the constants */
            dqSetKVal(oAu00,AU_L_K1,K);
            dqSetKVal(oAu00,AU_L_K0,OneMinusK);
        }
        while(1);

        dqDisposeSys(oSystem);
}



/* auxiliary functions to set up the AM gateways to the correct speed */
amSetRcvGateway20MHz(oAmDev)
DqIPDev oAmDev;
{
    dqConnect(oAmDev,AM_SPUP0,AM_INPUT0);
    dqConnect(oAmDev,AM_INPUT0,AM_OP0);
    dqSpecLogic(oAmDev,AM_LOGIC4,0,1);
    dqSpecLogic(oAmDev,AM_LOGIC0,0xffffffff,0);
    amSetGateSysClkMult(oAmDev,AM_RCV,2);
}


amSetXmtGateway20MHz(oAmDev)
DqIPDev oAmDev;
{
    amSetGateSysClkMult(oAmDev,AM_XMT,2);
    dqConnect(oAmDev,AM_XMT_OUT,AM_OP3);
    dqSpecLogic(oAmDev,AM_LOGIC3,0xffffffff,0);
    dqConnect(oAmDev,AM_SLDN0_T0,AM_OUTPUT0);
    dqConnect(oAmDev,AM_XMT,AM_XMT_OUT);
}


amSetDispGateway40MHz(oAmDev)
DqIPDev oAmDev;
{
    amSetGateSysClkMult(oAmDev,AM_XMT,4);
    dqConnect(oAmDev,AM_XMT_OUT,AM_OP3);
    dqSpecLogic(oAmDev,AM_LOGIC3,0xff,0);
    dqConnect(oAmDev,AM_LOGIC3,AM_OUTPUT0);
    dqConnect(oAmDev,AM_XMT,AM_XMT_OUT);
}
```

## mosc.c

```
/*************************************************************************

    mosc.c - time average a video stream in a central fovea area

    This program will apply a recursive filter to time average the video
    data.  The filter that will be applied is:

      y(n) = K*x(n) + (1 - K)*y(n - 1)

      where K is a constant between 0 and 1
            y(n) is the current averaged frame
            y(n-1) is the previous averaged frame
            x(n) is the current input frame

    The area of the image where the filter will be applied is controlled
    by a MOSC element from a mask in memory.

*************************************************************************/

#include <stdio.h>
#include <datacube.h>

/* define the display size */
#define DISPLAYXSIZE    512L
#define DISPLAYYSIZE    484L

/* DC is here only because it is in the chasis as SPC master */
dqLimitIPDevSet(AB AM AS AG AU DC);



main()
{
    DqSystem oSystem;
    DqIPDev oAb00, oAm00, oAm01, oAm02, oAm05, oAg00, oAs00, oAu00;
    DqSurf oDispSrcSurf, oDispDstSurf, oAcqSrcSurf, oAcqDstSurf;
    DqSurf oProcSrcSurf,oProcDstSurfs[3];
    DqSurf FoveaMask, LastFrameSrc;
    DqPipe oDispPipe, oAcqPipe, oProcPipe;

    /* initialize the hardware and ImageFlow software */
    dqInitEnv();

    /* get handles for the system and all devices being used */
    oSystem = dqCreateStdSys();
    oAb00 = dqFindIPDev(oSystem, "ab00");
    oAs00 = dqFindIPDev(oSystem, "as00");
    oAg00 = dqFindIPDev(oSystem, "ag00");
    oAu00 = dqFindIPDev(oSystem, "au00");

    oAm00 = dqFindIPDev(oSystem, "ab00:am00");
    oAm01 = dqFindIPDev(oSystem, "ab00:am01");
    oAm02 = dqFindIPDev(oSystem, "ab00:am02");
    oAm05 = dqFindIPDev(oSystem, "ab00:am05");

  /* set the data type checking to tolerant */
    dqSetDTPDogma(DQ_DTP_TOLERANT);


    /********************  ACQUISITION PIPE ****************************/
    oAcqSrcSurf = dqCreateStdSizeSurf(oAs00, AS_ADC);
    dqAttachSurf(oAcqSrcSurf, AS_XMT);

    dqConnect(oAb00, DQ_CSG, AB_OP05);      /* use unsigned output of AS */

amSetRcvGateway20MHz(oAm05);

oAcqDstSurf = dqCreateSameSizeSurf(oAm05, AM_MEM_R8,oAcqSrcSurf);
dqAttachSurf(oAcqDstSurf,AM_RCV);

/* create a continuously running pipe, arm and fire it */
oAcqPipe = dqCreatePipe(oAcqDstSurf, DQ_TRG_CONTINUOUS);
dqArmPipe(oAcqPipe, DQ_DSM_PIPE);
dqFirePipe(oAcqPipe);


/********************** PROCESSING PIPE ****************************/
/* start the video side of the process pipe with a dup of the
   acquisition pipe destination */
oProcSrcSurf = dqDupSurf(oAcqDstSurf);
dqAttachSurf(oProcSrcSurf, AM_XMT);
amSetXmtGateway20MHz(oAm05);

dqSetSurfBaseDT(oProcSrcSurf,DQ_DT_UNSIGNED);

/* across the cross point to the time filter */
dqConnect(oAb00,DQ_IMX5,AB_OP08);        /* current frame into Moe */
dqSetKVal(oAb00,AB_OP09,0);
dqConnect(oAu00,AU_I_SEP_MOE,AU_I_MOE);
dqConnect(oAu00,AU_I_MOE_X0,AU_G_OP8);

/* create K * x(n)  K set in AU_L_K1 */
dqConnect(oAu00,AU_L_EXT0,AU_L_MULT1_BOP);
dqSetKBaseDT(oAu00,AU_L_K1,DQ_DT_SIGNED);
dqConnect(oAu00,AU_L_K1,AU_L_MULT1_AOP);

/* create surface to save the last frame */
LastFrameSrc = dqCreateSameSizeSurf(oAm01,AM_MEM_R8,oProcSrcSurf);
dqSetSurfBaseDT(LastFrameSrc,DQ_DT_UNSIGNED);
dqAttachSurf(LastFrameSrc,AM_XMT);
amSetXmtGateway20MHz(oAm01);

/* bring last frame across the cross point to the time filter */
dqConnect(oAb00,DQ_IMX1,AB_OP10);        /* last frame into Larry */
dqSetKVal(oAb00,AB_OP11,0);
dqConnect(oAu00,AU_I_SEP_LARRY,AU_I_LARRY);
dqConnect(oAu00,AU_I_LARRY_X0,AU_G_OP9);

/* create (1 - K) * y(n - 1)  (1 - K) set in AU_L_K0 */
dqConnect(oAu00,AU_L_EXT1,AU_L_MULT0_AOP);
dqSetKBaseDT(oAu00,AU_L_K0,DQ_DT_SIGNED);
dqConnect(oAu00,AU_L_K0,AU_L_MULT0_BOP);

/* zero down the AU_L_MULT2 and AU_L_MULT3 chains */
dqSetKVal(oAu00,AU_L_K2,0x00);
dqConnect(oAu00,AU_L_K2,AU_L_MULT2_AOP);
dqConnect(oAu00,AU_L_K2,AU_L_MULT2_BOP);
dqSetKVal(oAu00,AU_L_K3,0x00);
dqConnect(oAu00,AU_L_K3,AU_L_MULT3_AOP);
dqConnect(oAu00,AU_L_K3,AU_L_MULT3_BOP);

/*
  Combine to get K*x(n) + (1 - K)*y(n - 1)
  Calculations are performed with binary point at bit 7-8
  This shift moves binary point to bit 0-1 when leaving AU_L_SHIFT3
*/
dqSpecShift(oAu00,AU_L_SHIFT3,DQ_SHIFT_ARITHMETIC,-7);

/* Then effect rounding by adding 1 before AU_L_ADD3 tosses the LSB */
dqSetKVal(oAu00,AU_L_K4,1);
```

```
dqConnect(oAu00,AU_L_K4,AU_L_ADD3_BOP);

/* now straight shot to AU_L_RESULT */
dqConnect(oAu00,AU_L_CLIP,AU_L_P_RES);
dqConnect(oAu00,AU_L_P_RES,AU_L_RESULT);

/* pass result back through the AU cross point */
dqConnect(oAu00,AU_L_RESULT_X0,AU_G_OP1);
dqConnect(oAu00,AU_O_CLIP1,AU_O_C7_SRC);

/* across cross point to display and last frame surface */
dqConnect(oAb00,DQ_CU07,AB_OP01);
dqConnect(oAb00,DQ_CU07,AB_OP00);

/* create surface to save this frame for next time */
oProcDstSurfs[0] = dqDupSurf(LastFrameSrc);
dqAttachSurf(oProcDstSurfs[0],AM_RCV);
amSetRcvGateway20MHz(oAm01);

/* create surface for the display output */
oProcDstSurfs[1] = dqCreateSameSizeSurf(oAm00,AM_MEM_R8,oProcSrcSurf);
dqAttachSurf(oProcDstSurfs[1],AM_RCV);
amSetRcvGateway20MHz(oAm00);

/*********************************************************************/
/*                                                                   */
/*  BEGIN ADDED CODE FOR MOSC CONTROL                                */
/*                                                                   */
/*********************************************************************/

/* create the surface for the fovea mask image */
FoveaMask = dqCreateSameSizeSurf(oAm02,AM_MEM_R8,oProcSrcSurf);
dqSetSurfBaseDT(FoveaMask,DQ_DT_UNSIGNED);
dqAttachSurf(FoveaMask,AM_XMT);
amSetXmtGateway20MHz(oAm02);

/* fovea mask across the cross point to AU binary input section */
dqConnect(oAb00,DQ_IMX2,AB_OP12);
dqConnect(oAu00,DQ_CU04_D0,AU_I_BIN0_PRE);
dqConnect(oAu00,AU_I_BIN0_PRE,AU_I_BIN0);

/* through binary cross point to control AU_L_MOSC0 + 1 */
dqConnect(oAu00,AU_I_BIN0,AU_B_OP3);

/*********************************************************************/
/*                                                                   */
/*  END ADDED CODE FOR MOSC CONTROL                                  */
/*                                                                   */
/*********************************************************************/

/* terminate surface list and create pipe */
oProcDstSurfs[2] = 0;
oProcPipe = dqCreateMultiDstPipe(oProcDstSurfs,DQ_TRG_CONTINUOUS);
dqArmPipe(oProcPipe,DQ_DSM_PIPE);
dqFirePipe(oProcPipe);


/********************** DISPLAY PIPE ******************************/

oDispSrcSurf = dqDupSurf(oProcDstSurfs[1]);
dqAttachSurf(oDispSrcSurf,AM_XMT);
dqSpecXmtExpansion(oAm00,AM_XMT,4,1);

amSetDispGateway40MHz(oAm00);

dqConnect(oAg00, AG_RED, AG_DAC_LUT_SRC);
dqConnect(oAg00, AG_DAC_LUT, AG_DAC_SRC);
oDispDstSurf = dqCreateSurf(oAg00, AG_DAC,DISPLAYXSIZE,DISPLAYYSIZE);
dqAttachSurfGate(oDispDstSurf, AG_RCV);

dqSpecRcvShrinkage(oAg00,AG_RCV,4,1);

/* create the display pipe, arm and fire it */
oDispPipe = dqCreatePipe(oDispDstSurf, DQ_TRG_CONTINUOUS);
dqArmPipe(oDispPipe, DQ_DSM_PIPE);
dqFirePipe(oDispPipe);


/* loop for input of K and Radius values */
do
{
    float NewK;
    int K,OneMinusK;
    int Radius;

    printf("Enter K between 0.0 and 1.0 (<0 to stop): ");
    scanf("%f",&NewK);
    if(NewK < 0.0)
        break;
    if(NewK > 1.0)
        continue;

    K = irint(256.0 * NewK);
    OneMinusK = irint(256.0 * (1.0 - NewK));

/*********************************************************************/
/*                                                                   */
/*  BEGIN ADDED CODE FOR MOSC CONTROL                                */
/*                                                                   */
/*********************************************************************/

    printf("Enter radius of central area: ");
    scanf("%d",&Radius);

    /* stop the pipe to set these values */
    dqHaltPipe(oProcPipe);

    /* fill the mask with 0's and center region with 1's */
    gsClearView(FoveaMask,0x00);
    gsSetFillStyle(FoveaMask,GS_SOLID);
    gsSetFillColor(FoveaMask,0x01);
    gsCircle(FoveaMask,DISPLAYXSIZE/2,DISPLAYYSIZE/2,Radius);


    /* MOSC state 1 => fovea; state 0 = periphery */

    /* AU_L_MOSC1 controls k * x(n) */
    dqSetMoscState(oAu00,AU_L_MOSC1,1);
    dqSetKVal(oAu00,AU_L_K1,K);

    dqSetMoscState(oAu00,AU_L_MOSC1,0);
    dqSetKVal(oAu00,AU_L_K1,256);

    /* AU_L_MOSC0 controls (1 - K) * y(n - 1) */
    dqSetMoscState(oAu00,AU_L_MOSC0,1);
    dqSetKVal(oAu00,AU_L_K0,OneMinusK);

    dqSetMoscState(oAu00,AU_L_MOSC0,0);
    dqSetKVal(oAu00,AU_L_K0,0);
```

**mosc.c**

```
        /* get the pipe running again.
            No changes made to topology or delays */
        dqArmPipe(oProcPipe,DQ_DSM_NONE);
        dqFirePipe(oProcPipe);

    /*********************************************************************/
    /*                                                                   */
    /*  END ADDED CODE FOR MOSC CONTROL                                  */
    /*                                                                   */
    /*********************************************************************/

    }
    while(1);

    dqDisposeSys(oSystem);
}



/* auxiliary functions to set up the AM gateways to the correct speed */
amSetRcvGateway20MHz(oAmDev)
DqIPDev oAmDev;
{
    dqConnect(oAmDev,AM_SPUP0,AM_INPUT0);
    dqConnect(oAmDev,AM_INPUT0,AM_OP0);
    dqSpecLogic(oAmDev,AM_LOGIC4,0,1);
    dqSpecLogic(oAmDev,AM_LOGIC0,0xffffffff,0);
    amSetGateSysClkMult(oAmDev,AM_RCV,2);
}


amSetXmtGateway20MHz(oAmDev)
DqIPDev oAmDev;
{
    amSetGateSysClkMult(oAmDev,AM_XMT,2);
    dqConnect(oAmDev,AM_XMT_OUT,AM_OP3);
    dqSpecLogic(oAmDev,AM_LOGIC3,0xffffffff,0);
    dqConnect(oAmDev,AM_SLDN0_T0,AM_OUTPUT0);
    dqConnect(oAmDev,AM_XMT,AM_XMT_OUT);
}


amSetDispGateway40MHz(oAmDev)
DqIPDev oAmDev;
{
    amSetGateSysClkMult(oAmDev,AM_XMT,4);
    dqConnect(oAmDev,AM_XMT_OUT,AM_OP3);
    dqSpecLogic(oAmDev,AM_LOGIC3,0xff,0);
    dqConnect(oAmDev,AM_LOGIC3,AM_OUTPUT0);
    dqConnect(oAmDev,AM_XMT,AM_XMT_OUT);
}
```

## multiops.c

```
/*************************************************************************

    multiops.c - perform multiple operations on a video signal

  This program will perform several functions on a video signal.  They
  will be: unmodified, 3x3 convolution for horizontal edges, 3x3
  convolution for vertical edges, recursive frame averaging filter.

  The results of these four operations will be displayed simultaneously
  one a 1kx1k monitor.

  The program demonstrates the use of Pat's to perform fast pipe
  modifications.

*************************************************************************/

#include <stdio.h>
#include <datacube.h>

/* this must use a monitor capable of displaying a 1kx1k image */
#define DISPLAYXSIZE    1024L
#define DISPLAYYSIZE    1024L

/* convolution pattern */
#define KERNELXSIZE 3
#define KERNELYSIZE 3


/* identify the devices used in this program */
/* DC is here only because it is in the chasis as SPC master */
dqLimitIPDevSet(AB AM AS AG AP AU DC);

/* the size of the video frames that are being processed */
DqRect AcqRect = {0,0,511,483};

main()
{
    DqSystem oSystem;                                /* the system */
    DqIPDev oAb00, oAg00, oAs00, oAp00, oAu00; /* the devices */
    DqIPDev oAm00, oAm01, oAm02, oAm03;

  /* the memory surfaces */
    DqSurf AcqSrcSurf, AcqDstSurfs[3];
    DqSurf ConvSrcSurf, ConvDstSurf;
  DqSurf HKernel,VKernel;
  DqSurf AverageSrcSurf,LastFrameSrc,AverageDstSurfs[3];
    DqSurf DispSrcSurf, DispDstSurf;

  /* the pipes and their events */
    DqPipe DispPipe, AcqPipe, ConvPipe, AveragePipe;
  int AcqEvent,ConvEvent;

  DqByte HCoefs[3][3];        /* horizontal edge kernel coefficients */
  DqByte VCoefs[3][3];        /* vertical kernel coefficients */
    DqRect tConvRect;             /* used to specify the size of the kernal */

  /* the PAT's and their events */
  int NormalPat,VertPat,HorizPat,AveragePat;
  int NormalEvent,VertEvent,HorizEvent,AverageEvent,AveragePatEvent;

  /* the frame averaging filter coefficients */
    float NewK;
    int K,OneMinusK;

    char UserInpBuf[80];

    /* initialize the system and get handles for the devices */
    dqInitEnv();

    oSystem = dqCreateStdSys();

    oAb00 = dqFindIPDev(oSystem, "ab00");
    oAs00 = dqFindIPDev(oSystem, "as00");
    oAg00 = dqFindIPDev(oSystem, "ag00");
    oAp00 = dqFindIPDev(oSystem, "ap00");
    oAu00 = dqFindIPDev(oSystem, "au00");

    oAm00 = dqFindIPDev(oSystem, "ab00:am00");
    oAm01 = dqFindIPDev(oSystem, "ab00:am01");
    oAm02 = dqFindIPDev(oSystem, "ab00:am02");
    oAm03 = dqFindIPDev(oSystem, "ab00:am03");

  /* set the data type checking to tolerant */
    dqSetDTPDogma(DQ_DTP_TOLERANT);

    /************************ ACQUISITION PIPE ************************/
    AcqSrcSurf = dqCreateStdSizeSurf(oAs00, AS_ADC);
    dqAttachSurf(AcqSrcSurf, AS_XMT);

    /* take signed data from the AS device for the convolution */
    dqSetDTT(oAs00, AS_DTM, DQ_DTT_X_SIGNED);

    /* use DQ_CSR for signed output from the AS device */
    dqConnect(oAb00, DQ_CSR, AB_OP00);
    amSetRcvGateway20MHz(oAm00);
    AcqDstSurfs[0] = dqCreateSameSizeSurf(oAm00, AM_MEM_R8,AcqSrcSurf);
    dqAttachSurf(AcqDstSurfs[0],AM_RCV);

  /* use DQ_CSG for unsigned AS data for the frame averaging */
  dqConnect(oAb00,DQ_CSG,AB_OP02);
  amSetRcvGateway20MHz(oAm02);
  AcqDstSurfs[1] = dqCreateSameSizeSurf(oAm02, AM_MEM_R8,AcqSrcSurf);
    dqAttachSurf(AcqDstSurfs[1],AM_RCV);

    /* create a continuously running multi-destination pipe,
     arm and fire it off */
  AcqDstSurfs[2] = 0;                    /* terminate multiple surface list */
    AcqPipe = dqCreateMultiDstPipe(AcqDstSurfs, DQ_TRG_CONTINUOUS);
    dqArmPipe(AcqPipe, DQ_DSM_PIPE);
    dqFirePipe(AcqPipe);
  AcqEvent = emFindPipeEvent(AcqPipe);

    /********************** DISPLAY PIPE ******************************/
  /* This pipe will be 1024x1024. It will not work on an NTSC monitor!! */
    DispSrcSurf = dqCreateSurf(oAm01, AM_MEM_R8,DISPLAYXSIZE,DISPLAYYSIZE);
  gsClearView(DispSrcSurf,0);                                  /* clear the surface */
    dqAttachSurf(DispSrcSurf,AM_XMT);

    amSetDispGateway40MHz(oAm01);
  /* Because this is a higher resolution monitor the dummy expansion and
     shrinkage factors are different */
    dqSpecXmtExpansion(oAm01,AM_XMT,1,1);

    dqConnect(oAg00, AG_GREEN, AG_DAC_LUT_SRC);
    dqConnect(oAg00, AG_DAC_LUT, AG_DAC_SRC);

    DispDstSurf = dqCreateSurf(oAg00, AG_DAC,DISPLAYXSIZE,DISPLAYYSIZE);
    dqAttachSurfGate(DispDstSurf, AG_RCV);
    dqSpecRcvShrinkage(oAg00,AG_RCV,1,1);

    DispPipe = dqCreatePipe(DispDstSurf, DQ_TRG_CONTINUOUS);
```

## multiops.c

```
  dqArmPipe(DispPipe, DQ_DSM_PIPE);
  dqFirePipe(DispPipe);


/********************** PROCESSING PIPES ****************************/

/******** CONVOLVED IMAGE *********/
  /* start with the signed data from the acquisition pipe */
  ConvSrcSurf = dqDupSurf(AcqDstSurfs[0]);
  dqAttachSurf(ConvSrcSurf, AM_XMT);

  /* because this is running into the cross point for processing the
      memory must output at 20 MHz. */
  amSetXmtGateway20MHz(oAm00);

  dqConnect(oAb00, DQ_IMX0, AB_OP21);     /* across to the NMAC input */

  /* set the configuration for the NMAC element to 8x8 convolution */
  dqConnect(oAp00, AP_SHIFT8, AP_NDLY_SRC);

  /* the host will write directly into the surface with the coeffient
      values.  This rectangle is defined with respect to center point.
      The default center point is the center pixel of the kernel. */
  tConvRect.lXMin = -(KERNELXSIZE / 2);
  tConvRect.lXMax = KERNELXSIZE % 2 ? KERNELXSIZE / 2
                                    : KERNELXSIZE / 2 - 1;
  tConvRect.lYMin = -(KERNELYSIZE / 2);
  tConvRect.lYMax = KERNELYSIZE % 2 ? KERNELYSIZE / 2
                                    : KERNELYSIZE / 2 - 1;

/* We can define multiple kernels in the 8x8 convolution surface.  They
   are defined as surfaces in the NMAC element.  Only one of them can
   be attached, ie. in use, at a time though. */
/******* HORIZONTAL EDGE KERNEL ***********/
  /* create a surface to define the horizontal kernel in the NMAC */
  HKernel = dqCreateSurf(oAp00, AP_NMAC8, KERNELXSIZE, KERNELYSIZE);
  dqSetSurfBaseDT(HKernel, DQ_DT_SIGNED); /* everything will be signed */
  dqAttachSurf(HKernel, AP_NMAC8);    /* you have to attach to this also */

  /* this defines a 3x3 horizontal line filter */
  HCoefs[0][0] = 64;
  HCoefs[0][1] = 64;
  HCoefs[0][2] = 64;
  HCoefs[1][0] = 0;
  HCoefs[1][1] = 0;
  HCoefs[1][2] = 0;
  HCoefs[2][0] = -64;
  HCoefs[2][1] = -64;
  HCoefs[2][2] = -64;

  dqWtRect(HKernel, &tConvRect, HCoefs);


/******* VERTICAL EDGE KERNEL ***********/
  /* create a surface to define the vertical kernel in the NMAC */
  VKernel = dqCreateSurf(oAp00, AP_NMAC8, KERNELXSIZE, KERNELYSIZE);
  dqSetSurfBaseDT(VKernel, DQ_DT_SIGNED); /* everything will be signed */
  dqAttachSurf(VKernel, AP_NMAC8);   /* you have to attach to this also */

  /* this defines a 3x3 horizontal line filter */
  VCoefs[0][0] = 64;
  VCoefs[0][1] = 0;
  VCoefs[0][2] = -64;
  VCoefs[1][0] = 64;
  VCoefs[1][1] = 0;
  VCoefs[1][2] = -64;
  VCoefs[2][0] = 64;
  VCoefs[2][1] = 0;
  VCoefs[2][2] = -64;

  dqWtRect(VKernel, &tConvRect, VCoefs);

  /* define constants that feed the AP_ADD B operand */
  dqSetKVal(oAb00,AB_OP17,0L);
  dqSetKVal(oAb00,AB_OP18,0L);
  dqSetKVal(oAb00,AB_OP19,0L);
dqSetOpndBaseDT(oAp00,AP_ADD,DQ_OPND_A,DQ_DT_SIGNED);

  /* shift so that the 8-bit result comes out at DQ_CP15.  This
      shifting must also be accounted for when setting the NMAC
      coefficients */
  dqSpecShift(oAp00, AP_SHIFT8, DQ_SHIFT_ARITHMETIC, 17);
  dqSpecShift(oAp00, AP_SHIFT,  DQ_SHIFT_ARITHMETIC,  7);

/* now set the output format to account for the convolution.
   Horizontal edges are convolved toward -127 and 127. Use the
   absolute value output so that both edges are driven toward
   127.  This also shifts the values up one place. */
dqSetOpndBaseDT(oAp00,AP_FORMAT,DQ_OPND_A,DQ_DT_SIGNED);
dqSetFormatOp(oAp00,AP_FORMAT,AP_FMT_ABSOLUTE);

  amSetRcvGateway20MHz(oAm01);

/* deposit image in a duplicate of the display source surface */
  ConvDstSurf = dqDupSurf(DispSrcSurf);

gsClearView(ConvDstSurf,0);                                    /* clear the screen */

/* the rest of the convolution pipe definition is in the Pat's */



/********* AVERAGING PIPE ********************/
/* start with the unsigned data from the acquisition pipe */
AverageSrcSurf = dqDupSurf(AcqDstSurfs[1]);
  dqAttachSurf(AverageSrcSurf, AM_XMT);
  amSetXmtGateway20MHz(oAm02);

  dqSetSurfBaseDT(AverageSrcSurf,DQ_DT_UNSIGNED);

  /* across the cross point to the time filter */
  dqConnect(oAb00,DQ_IMX2,AB_OP08);        /* current frame into Moe */
  dqSetKVal(oAb00,AB_OP09,0);
  dqConnect(oAu00,AU_I_SEP_MOE,AU_I_MOE);
  dqConnect(oAu00,AU_I_MOE_X0,AU_G_OP8);

  /* create K * x(n)  K set in AU_L_K1 */
  dqConnect(oAu00,AU_L_EXT0,AU_L_MULT1_BOP);
  dqSetKBaseDT(oAu00,AU_L_K1,DQ_DT_SIGNED);
  dqConnect(oAu00,AU_L_K1,AU_L_MULT1_AOP);

  /* create surface to save the last frame */
  LastFrameSrc = dqCreateSameSizeSurf(oAm03,AM_MEM_R8,AverageSrcSurf);
  dqSetSurfBaseDT(LastFrameSrc,DQ_DT_UNSIGNED);
  dqAttachSurf(LastFrameSrc,AM_XMT);
  amSetXmtGateway20MHz(oAm03);

  /* bring last frame across the cross point to the time filter */
  dqConnect(oAb00,DQ_IMX3,AB_OP10);        /* last frame into Larry */
  dqSetKVal(oAb00,AB_OP11,0);
```

```
  dqConnect(oAu00,AU_I_SEP_LARRY,AU_I_LARRY);
  dqConnect(oAu00,AU_I_LARRY_X0,AU_G_OP9);

  /* create (1 - K) * y(n - 1)  (1 - K) set in AU_L_K0 */
  dqConnect(oAu00,AU_L_EXT1,AU_L_MULT0_AOP);
  dqSetKBaseDT(oAu00,AU_L_K0,DQ_DT_SIGNED);
  dqConnect(oAu00,AU_L_K0,AU_L_MULT0_BOP);

  /* zero down the AU_L_MULT2 and AU_L_MULT3 chains */
  dqSetKVal(oAu00,AU_L_K2,0x00);
  dqConnect(oAu00,AU_L_K2,AU_L_MULT2_AOP);
  dqConnect(oAu00,AU_L_K2,AU_L_MULT2_BOP);
  dqSetKVal(oAu00,AU_L_K3,0x00);
  dqConnect(oAu00,AU_L_K3,AU_L_MULT3_AOP);
  dqConnect(oAu00,AU_L_K3,AU_L_MULT3_BOP);

  /*
    Combine to get K*x(n) + (1 - K)*y(n - 1)
    Calculations are performed with binary point at bit 7-8
    This shift moves binary point to bit 0-1 when leaving AU_L_SHIFT3
  */
  dqSpecShift(oAu00,AU_L_SHIFT3,DQ_SHIFT_ARITHMETIC,-7);

  /* Then effect rounding by adding 1 before AU_L_ADD3 tosses the LSB */
  dqSetKVal(oAu00,AU_L_K4,1);
  dqConnect(oAu00,AU_L_K4,AU_L_ADD3_BOP);

  /* now straight shot to AU_L_RESULT */
  dqConnect(oAu00,AU_L_CLIP,AU_L_P_RES);
  dqConnect(oAu00,AU_L_P_RES,AU_L_RESULT);

  /* pass result back through the AU cross point */
  dqConnect(oAu00,AU_L_RESULT_X0,AU_G_OP1);
  dqConnect(oAu00,AU_O_CLIP1,AU_O_C7_SRC);

  /* across cross point to the last frame surface */
  dqConnect(oAb00,DQ_CU07,AB_OP03);

  /* create surface to save this frame for next time */
  AverageDstSurfs[0] = dqDupSurf(LastFrameSrc);
  dqAttachSurf(AverageDstSurfs[0],AM_RCV);
  amSetRcvGateway20MHz(oAm03);

  /* Create surface for the display output.  This is a second duplicate of
   the display source surface.  It will be used for writting the
   unmodified and averaged images. */
  AverageDstSurfs[1] = dqDupSurf(DispSrcSurf);
  dqAttachSurf(AverageDstSurfs[1],AM_RCV);

/* the receive gateway on oAM01 was set up as part of the convolution
   pipe.  The rest of the averaging pipe is set up in the pat's */

 /* terminate surface list and create pipe */
  AverageDstSurfs[2] = 0;

/* get the averaging constant to use */
do
  {
      printf("Enter K between 0.0 and 1.0: ");
      scanf("%f",&NewK);
}
while((NewK < 0) || (NewK > 1.0));

  K = irint(256.0 * NewK);
  OneMinusK = irint(256.0 * (1.0 - NewK));
  dqSetKVal(oAu00,AU_L_K1,K);
  dqSetKVal(oAu00,AU_L_K0,OneMinusK);



/************** PAT DEFINITIONS ***************/


/***** HORIZONTAL EDGES ***********/
emBegPatDef();
dqAttachSurf(HKernel, AP_NMAC8);                /* attach horizontal kernel */
  dqConnect(oAb00, DQ_CP15, AB_OP01);           /* connect AP output to display */

/* attach the duplicated surface that is the convolution pipe
   destination. */
  dqAttachSurf(ConvDstSurf,AM_RCV);
dqSpecSurfAlignPoint(ConvDstSurf,0,512); /* lower left corner */
dqSetSurfProcRect(ConvDstSurf,AcqRect);

/* This pipe creation is done only once.  It is not a deferred operation.
   The purpose within Imageflow is to create an identification for the
   pipe that can be used by other functions. */
  ConvPipe = dqCreatePipe(ConvDstSurf, DQ_TRG_ONESHOT);

/* For one-shot pipes we must get a pipe event after each arming */
dqArmPipe(ConvPipe,DQ_DSM_PIPE);
ConvEvent = emFindPipeEvent(ConvPipe);
dqFirePipe(ConvPipe);

emWaitRefEvent(ConvEvent,1);           /* wait for pipe operation to complete */

/* end of the pat definition and getting an event handle for it */
HorizPat = emEndPatDef();
HorizEvent = emFindPatEvent(HorizPat);


/***** VERTICAL EDGES *************/
emBegPatDef();
dqAttachSurf(VKernel,AP_NMAC8);                             /* vertical edge kernel */

dqSpecSurfAlignPoint(ConvDstSurf,512,512); /* lower right corner */
dqSetSurfProcRect(ConvDstSurf,AcqRect);

  dqArmPipe(ConvPipe, DQ_DSM_RECT);
ConvEvent = emFindPipeEvent(ConvPipe);
  dqFirePipe(ConvPipe);

emWaitRefEvent(ConvEvent,1);

VertPat = emEndPatDef();
VertEvent = emFindPatEvent(VertPat);


/***** AVERAGED IMAGE **************/
emBegPatDef();

/* connect the output of the AU which is the averaged frame */
  dqConnect(oAb00,DQ_CU07,AB_OP01);

/* attach the duplicated surface that is the destination for the frame
   averaging pipe */
dqAttachSurf(AverageDstSurfs[1],AM_RCV);
dqSpecSurfAlignPoint(AverageDstSurfs[1],512,0);
dqSetSurfProcRect(AverageDstSurfs[1],AcqRect);
```

## multiops.c

```
  /* again this is not a defferd operation.  It creates an identification
     for the pipe. */
    AveragePipe = dqCreateMultiDstPipe(AverageDstSurfs,DQ_TRG_ONESHOT);
    dqArmPipe(AveragePipe,DQ_DSM_PIPE);
  AverageEvent = emFindPipeEvent(AveragePipe);
    dqFirePipe(AveragePipe);

  emWaitRefEvent(AverageEvent,1);

  AveragePat = emEndPatDef();
  AveragePatEvent = emFindPatEvent(AveragePat);


  /***** NORMAL IMAGE ***************/
  emBegPatDef();

  dqConnect(oAb00,DQ_IMX2,AB_OP01);               /* normal video to proc dest */

  dqSpecSurfAlignPoint(AverageDstSurfs[1],0,0);
  dqSetSurfProcRect(AverageDstSurfs[1],AcqRect);

  dqArmPipe(AveragePipe,DQ_DSM_PIPE);
  AverageEvent = emFindPipeEvent(AveragePipe);
  dqFirePipe(AveragePipe);

  emWaitRefEvent(AverageEvent,1);

  NormalPat = emEndPatDef();
  NormalEvent = emFindPatEvent(NormalPat);

  /********* END OF PAT DEFINITIONS *****************/

    /* create the loop of pat firings */
  emCyclePatOnEvent(HorizPat,NormalEvent);
  emCyclePatOnEvent(VertPat,HorizEvent);
  emCyclePatOnEvent(AveragePat,VertEvent);
  emCyclePatOnEvent(NormalPat,AveragePatEvent);

  /* simulate one of them to get things rolling */
  emSimulateEvent(NormalEvent);


  gets(UserInpBuf);                     /* clear a \n from previous scanf */
  printf("Hit any key to exit.\n");             /* wait for user to signal exit */
  fgets(UserInpBuf,80,stdin);

  /* idle all pat's before trying to dispose the system.  If you do not
     take this step the program will usually hang on exit waiting for
     one of these to fire. */
  emIdlePat(NormalPat);
  emIdlePat(AveragePat);
  emIdlePat(VertPat);
  emIdlePat(HorizPat);

    dqDisposeSys(oSystem);                                      /* clean before exiting
*/
}



/* auxiliary functions to set up the AM gateways to the correct speed */
amSetXmtGateway20MHz(oAmDev)
DqIPDev oAmDev;
{
    amSetGateSysClkMult(oAmDev,AM_XMT,2);
    dqConnect(oAmDev,AM_XMT_OUT,AM_OP3);
    dqSpecLogic(oAmDev,AM_LOGIC3,0xffffffff,0);
    dqConnect(oAmDev,AM_SLDN0_T0,AM_OUTPUT0);
    dqConnect(oAmDev,AM_XMT,AM_XMT_OUT);
}


amSetRcvGateway20MHz(oAmDev)
DqIPDev oAmDev;
{
    dqConnect(oAmDev,AM_SPUP0,AM_INPUT0);
    dqConnect(oAmDev,AM_INPUT0,AM_OP0);
    dqSpecLogic(oAmDev,AM_LOGIC4,0,1);
    dqSpecLogic(oAmDev,AM_LOGIC0,0xffffffff,0);
    amSetGateSysClkMult(oAmDev,AM_RCV,2);
}


amSetDispGateway40MHz(oAmDev)
DqIPDev oAmDev;
{
    amSetGateSysClkMult(oAmDev,AM_XMT,4);
    dqConnect(oAmDev,AM_XMT_OUT,AM_OP3);
    dqSpecLogic(oAmDev,AM_LOGIC3,0xff,0);
    dqConnect(oAmDev,AM_LOGIC3,AM_OUTPUT0);
    dqConnect(oAmDev,AM_XMT,AM_XMT_OUT);
}
```