PPA Printer Firmware Design

Hewlett-Packard's new Printing Performance Architecture (PPA) includes a significantly reduced set of printer firmware. "Don't touch the dots" was the firmware designer's golden rule. This means that the firmware and processor do only mechanism control, I/O, command parsing, status reporting, user interface, and general housekeeping functions.

by Erik Kilk

A significant factor in Hewlett-Packard's new Printing Performance Architecture (see ***Article 1***) is the reduction of the processing power embedded in the printer. Using the host PC for all image formatting leaves only motor, print cartridge, I/O, user interface, command, and status functions to be controlled by the firmware. This results in significant cost savings by reducing processor needs and by reducing ROM and RAM requirements. The goal, which was achieved, was to reduce the ROM requirements to 64K bytes.

Fig. 1 shows the traditional firmware architecture used in HP DeskJet printers. The firmware receives from the host PC a combination of text, text formatting commands, and raster graphics data. This is formatted according to the Hewlett-Packard PCL printer language specification. The information to print arrives at a page description level, which requires firmware to rasterize a bit image, generate and place fonts, and format and cut the image into swaths according to the requirements and format of the print cartridge.

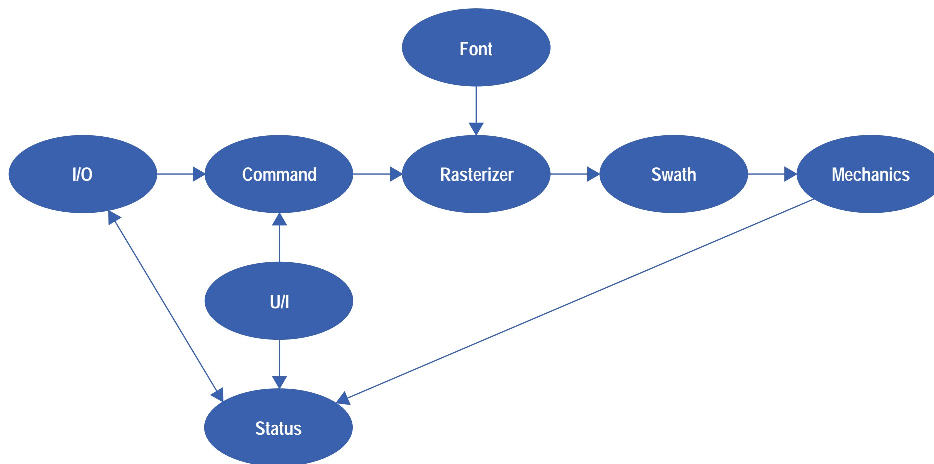

Fig. 1. *Traditional HP DeskJet printer firmware architecture.*

At the I/O layer, previous HP DeskJet printers make use of the Multiple Logical Channel packetizing layer (MLC, being proposed as IEEE standard 1284.4) to offer multiple connections between a host and a printer. PCL and an HP proprietary peripheral status language share the bidirectional parallel port.

The rasterizing step involves converting text and text formatting commands into a graphical bit image to be printed. Separate bit-image planes are created for each of the four ink colors: black, cyan, magenta, and yellow.

The swath cutting step involves cutting the bit image into print-cartridge-high swaths, performing image enhancements such as overlapping print sweeps, and adjusting the bit-image planes to the particular format required by the print cartridges used in the printers.

In general, not only does the traditional HP DeskJet firmware consist of more modules but the modules themselves are considerably more complex than with the new Printing Performance Architecture.

PPA Firmware Architecture Overview

The primary goal of the Printing Performance Architecture, or PPA, is to reduce the price of an HP DeskJet printer while maintaining or increasing print performance. The digital electronics portion of this savings is accomplished by reducing ROM, RAM, and microprocessor costs. ROM is reduced by moving the rasterization, font, and swath module functions onto the host's printer driver, and by using streamlined I/O and command language protocols. RAM is reduced by requiring only enough RAM for a worse-case print sweep plus spare RAM for firmware overhead. Microprocessor costs are held down by reducing the processing, in particular the data processing, required of the microprocessor. The "don't touch the dots" concept enabled the use of a low-cost 68000 processor.

Fig. 2 shows the firmware architecture of the HP DeskJet 820C. It consists of a small set of communicating modules. Each module is implemented with a few communicating processes and interrupt service routines. The processes communicate through the use of messages (see below).

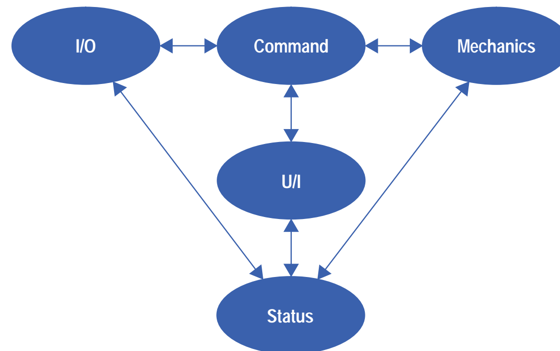

Fig. 2. *HP DeskJet 820C printer firmware architecture overview.*

The I/O module receives data and commands from the host PC, passing them on to the command module, and transmits responses and status information back to the PC. The command module parses and prioritizes the incoming commands and passes them on to the other modules, most often the mechanism module, for execution. The mechanism module receives paper load and eject, print sweep, and print cartridge servicing commands, performs the requested actions by controlling the motors and print cartridges, and passes the results back to the command module. The U/I (user interface) module handles the front-panel state machine, sending commands to the command module as necessary. The status module monitors the printer's status, communicates this status back to the PC via the I/O module, and keeps the rest of the modules informed of system status.

Processes, Messages, and Operating System

A small and efficient custom operating system manages the execution of multiple processes and the delivery of messages from one process to another. The operating system also provides support for interrupt service routines, delayed procedure calls, and binary semaphores.

Processes. Multiple, cooperating independent threads of execution called *processes* are used to provide priority, modularity, and parallelism within the PPA firmware architecture. Individual processes are instantiated with a function stack, a fixed priority of execution, and a specific set of broadcast classes. The highest priority ready process executes until either a higher-priority process becomes ready to execute, the current process is blocked waiting for a new message, or the process is blocked waiting for a semaphore to be unlocked. The process's broadcast classes indicate which set of broadcast messages the process wants to receive. Processes are static and never terminate.

A fundamental architectural concept is that there is a one-to-one correspondence between a process and a message queue. In other words, each and every process has its own queue for messages and no other queue. This concept is hardwired into the system. There are no facilities for the creation or use of any other message queues. When a process requests a message, its context defines which queue is selected.

The PPA firmware design is rather liberal with the use of processes to both modularize and parallelize functionality. Table I shows the eighteen processes used in the HP DeskJet 820C printer.

Table I
Firmware Processes in the HP DeskJet 820C Printer

Major Firmware Module					
I/O	**Command**	**Mechanism**	**Status**	**U/I**	**Other**
IO	Parser	Mechanism State Machine	Autostatus	UI	PState
IEEE 1284	Pacer	Walker/Dispatcher	Status Request		Configuration
VLink Pacing	Executer				NV RAM
					Execute Data
					Test Print
					Simple

Messages. Messages form the fundamental communication method between processes. Physically, messages are fixed-size, small blocks of memory. They contain both required and optional fields.

The typical life of a message is as follows. A process acquires an uninitialized message from the operating system. The process fills the necessary message fields. The message is posted to another process with a specific priority. The receiving process gets the message and performs the action implied by the message's identity. Depending on flags set within the message, a response message may be posted back to the originator or the message may be released back to the operating system for reuse.

The reception of messages can be gated by a priority or limited by a timeout or both. Messages can be posted to an individual process or broadcast to many processes. The posting of a message can be deferred for a specific time to provide for periodic actions. Interrupt service routines can only post messages, so arrangements must be made to acquire their messages outside of interrupt execution.

Table II shows the message structure. Messages include a token field, which gives the message an identity or specific meaning. For example, a command module process requests raw input data by posting to an I/O module process the RECV_REQUEST message (a message with its token set to RECV_REQUEST). A response field indicates which process is to be posted the result of the message. For example, when processing the RECV_REQUEST message, the I/O module process will post a response back to the process mentioned in the response field. A data pointer field, a size field, and a recover field associate a block of memory with a message. The recover field indicates which process is to be notified to recover the memory block when it is no longer needed. The use of associated data in this manner allows the firmware to pass data blocks from process to process and let the final process recover the data properly.

Semaphores. Semaphores provide a mechanism to restrict access to a shared resource (often global variables) to one process at a time. They are analogous to a lock on a door. Semaphores can be instantiated, locked, and unlocked. There are only a few critical uses of semaphores in the system. One is for the exclusive use of global configuration data. Another is for the exclusive use of the general-purpose memory pool.

Delayed Procedure Calls. Individual functions can be executed at a later time via the operating system. The operating system maintains a list of functions to be executed and at the appropriate time will execute the functions at a low interrupt level. Processes can take advantage of this feature to execute critical code at a higher-priority interrupt level. Interrupt service routines can take advantage of this feature to execute noncritical code at a lower interrupt level. Since a list of functions is maintained by the operating system, delayed procedure calls can be canceled. The user interface module uses deferred procedure calls to implement key debouncing. The deferred post feature of message posting is implemented by using deferred procedure calls.

Interrupt Service Routines. Interrupt routines are statically installed. In practice, interrupt routines often just post a message to wake up a process. For more sophisticated needs, interrupt routines can logically suspend until a subsequent interrupt. This facilitates designing serial and sequential interrupt state machines.

Memory Management. Memory management is strictly static with few exceptions. The operating system does not provide any sort of functionality to allocate or free memory. The reliability of the system was greatly enhanced by designing it for static memory use. The I/O module does provide for the use of its output ring buffer for general-purpose, restricted memory allocation with function calls such as Ring_Request() and Ring_Recover(). The restrictions were imposed for simplicity and because of the ring nature of the buffer: memory must be allocated in multiples of 4 bytes, memory must be held for a very short time or the efficiency of the output buffer will degrade, and although memory can be returned piecemeal, the pieces must start on 4-byte boundaries and be multiples of 4 bytes.

Table II
Message Structure

Message Field	Size	Description
Token	16 bits	Message identity. For example, RECV_REQUEST indicates this message is a request to receive data.
Sender	32 bits	Sending process's identity.
Response	32 bits	Identity of the process to receive the response to the message.
Data Pointer	32 bits	Pointer to an associated data block. For example, this could point to a block of input data for a RECV message.
Data Size	32 bits	Number of bytes of data associated with the message.
Recover	32 bits	Identity of the process to recover the associated data.
Flag	8 bits	Indicates whether the message must be responded to or data must be recovered. If a response message, indicates if failure, OK, or an unknown message type.
Misc 1	32 bits	Message-specific information.
Misc 2	32 bits	Message-specific information.

Firms (Soft Constants). A *firm* is a concept added to the firmware design to facilitate adjusting constants postrelease. Constants that may need adjustment after the printer has been released for manufacture are grouped together in lists. Access to these constants is via the Firm() function call. Firm() is called with a list and a constant identifier. Firm() looks up and returns the desired constant. Firm() also quickly scans a small constant replacement list. This replacement list includes the original list and constant identifier along with a new value for the constant. If a replacement exists, Firm() returns the replacement. The constant replacement list is stored in nonvolatile memory. Generally this would occur as a final step in the manufacturing process.

I/O Module

Fig. 3 shows how the I/O module is structured into physical, link, and application layers. The physical layer deals with the signaling on the parallel cable. The link layer deals with logically dividing a single cable into multiple logical channels. The application layer deals with the various data, command, status, and pacing applications necessary to implement the printer features.

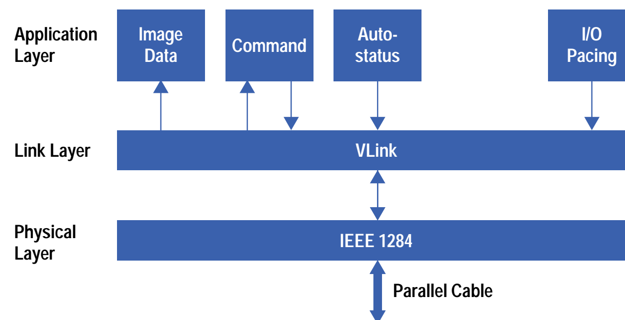

Fig. 3. *Layered I/O structure.*

Physical Layer—IEEE 1284 Parallel Port. The connector on the back of the HP DeskJet 820C printer connects to the parallel printer port on the PC. The IEEE 1284 bidirectional parallel port specification is supported by dedicated hardware and firmware. Hardware performs the basic data transfer of bytes from the PC directly into RAM. Firmware supports the IEEE 1284 overhead required to put the port in the proper transfer modes and to transfer data back to the PC.

IEEE 1284 redefines the traditional parallel port lines BUSY, NFAULT, PERR, and so on to permit faster data transmission and to allow data to be sent back to the PC from the printer. Faster data rates are achieved by having the host only pulsing the NSTROBE line until the printer raises its BUSY line. Traditionally the NSTROBE line had to be held down for a set minimum time period (which was a relatively long time).

The IEEE 1284 overhead for mode switching is implemented as a separate firmware process in the system. To achieve the IEEE 1284-required 35-ms response time, the process runs at the highest priority in the system. The process monitors the parallel port lines and responds to changes by maneuvering through a constant state table. This state table includes information on what to watch for on the parallel lines, how to respond on the parallel lines, how to get and retrieve data at the appropriate times, and which states can be expected next.

Link Layer—VLink. The link layer provides a simple logical channel protocol. To prevent the printer's input buffer from completely filling up and preventing communication with the PC, image data and command data are separated into two logical channels. Each of these two logical channels is individually paced to prevent one from blocking the other. To separate the data and commands into two logical channels, the raw bytes are packetized so that a channel number can be assigned to each packet.

A new HP-proprietary link-level protocol, VLink, replaces the more sophisticated MLC protocol used in the other DeskJet and LaserJet models. VLink requires considerably less code, can be substantially implemented in hardware, and doesn't require a bidirectional link.

Fig. 4 shows the VLink packets. To packetize the data, VLink adds four additional header bytes to each block of data. First, a start-of-packet character, $, is sent. Second, one byte specifying the channel number is sent. Third, a word is sent indicating the number of data bytes to follow. A packet can contain up to 64K bytes of data. Custom I/O hardware strips off the four header bytes, uses the channel number to select a ring buffer in RAM in which to store the data, and subsequently tranfers the data into the ring buffer by DMA.

Fig. 4. *VLink packet traveling on the physical cable.*

Table III shows how channels are allocated in the DeskJet 820C. Incoming packets arrive for either channel 0 or channel 1. Channel 0 is used for image data. Channel 1 is used for commands. Outgoing packets are transmitted using channels 1, 2, and 128. Outgoing channel 1 is used for responses to commands. Outgoing channel 2 is used for the periodic autostatus information. Outgoing channel 128 is used to supply pacing information to the host PC.

Ring Buffers. Two ring buffers store the two incoming data streams from the host PC. One stores the image data arriving on VLink channel 0. The other stores commands arriving on VLink channel 1. The ring buffers are implemented with a combination of custom hardware and firmware.

Table III
VLink Channel Uses

Use	Input Channel	Output Channel
Image Data	0	
Commands and Responses	1	1
Periodic Autostatus		2
Periodic Ring Buffer Pacing		128

Fig. 5 shows a diagram of a single ring buffer. The custom ASIC selects the ring buffer in which to deposit incoming data based upon the channel number in the VLink header. Incoming data bytes are placed into the byte pointed to by the ring's fill register, and the fill register is incremented. If the fill register passes the high wrap register, the fill register is set equal to the low wrap register. Once the fill register equals the recover register, no more input is permitted. Any further input for this ring buffer will cause the parallel port's BUSY line to be set high and remain high until the recover register is changed.

For the command ring buffer, the grant register (a firmware-only register not in the custom ASIC) is used to mark the data that has been granted to the parser. When the command associated with the data has completed executing, its data is recovered by advancing the recover register. This permits further data input.

For the image ring buffer, the ASIC advances the recover register as it pulls data out for the print cartridges. This occurs while the print sweep is taking place. This permits further input to occur on the image channel in case the buffer was previously full. The grant register does not exist for the image buffer.

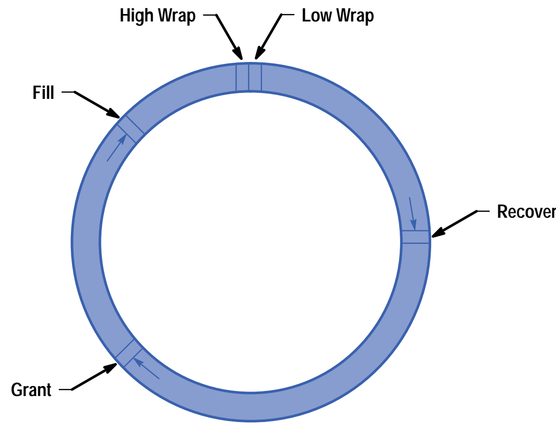

Fig. 5. *I/O ring buffer.*

A third ring buffer, one that is implemented entirely in firmware and has no custom ASIC registers, is used for an output buffer and occasionally general-purpose memory allocation. The same ring buffer design is used, thereby reusing the ring buffer utility functions. In this case, memory is granted to a process, advancing the grant register. Eventually the memory will be recovered, advancing the recover pointer. For this output buffer, the input register has no use.

An enhancement useful for both the command and general-purpose output ring buffers is the ability to recover blocks of memory out of order. This is facilitated by managing subrecovered blocks of memory between the grant pointer and the recover pointer. With the rule that all memory requests and recoveries must be restricted to multiples of 4 bytes, subrecovered blocks can be implemented using only the RAM contained within the recovered blocks themselves.

Output. For output, the main control I/O process receives SEND messages from the other processes within the system. Like input, output is formatted as VLink packets. Three VLink channels are used: channel 1 to transmit command responses back to the host, channel 2 to transmit periodic autostatus back to the host, and channel 128 to transmit I/O buffer pacing information back to the host.

The design handles cases when bidirectional I/O is not available. This can happen when the printer driver is busy and not communicating with the parallel port, when the driver is not running at all, when an external device using non-IEEE-1284-compliant cables is between the printer and the host, when the PC does not support IEEE 1284, or when there exist miscellaneous hardware and software conflicts with the parallel port.

In cases where bidirectional I/O is not available, output is prevented from accumulating inside the printer by buffering at most one packet per VLink channel. Any previous packets are automatically recovered back into the system and never transmitted. This priority scheme ensures that the host PC always receives the latest status. The only repercussion for bidirectional systems is that the driver cannot send multiple queries to the printer without waiting for each individual response.

Image Data. A key and early concept of PPA is that data arriving at the printer will already be formatted for the custom ASIC hardware controlling the print cartridges. In other words, the firmware and microprocessor in the printer do not process the data, nor do they move the data in RAM. The image data is transferred by DMA into the image ring buffer from the ASIC I/O block and from the ring buffer to the print control ASIC blocks.

Autostatus. To keep the host PC informed of the status of the printer, an autostatus process periodically formats a data block with the printer's current status. This data block is then given to the I/O module for transmission back to the host on VLink channel 2.

I/O Pacing. When one of the input ring buffers fills up completely and another byte arrives for this full ring buffer, the overflowing byte causes the parallel port's BUSY line to raise and hold off the host PC from transmitting any further data. Such a situation could prevent the host PC from querying the printer's status or canceling a print job, so the printer and host work together to prevent either of the input ring buffers from completely filling up, thus allowing the other ring buffer to continue to receive data.

The printer transmits back to the host periodic ring buffer status information on VLink channel 128. The data transmitted indicates both the instantaneous free space available in each buffer and the amount of data recovered from the ring buffers. The amount of data recovered from the ring buffers is cumulative. In other words, the printer reports the total number of bytes it has recovered from all of the input buffers.

This total number of recovered bytes permits the host PC to determine exactly how much space is available at any time in the printer's input buffers, as long as it keeps track of how many bytes it has itself transmitted. This mechanism is required because the printer's report of the free space available in the input buffer is only an instantaneous reading. It doesn't account for any data in transition and could thus give the host PC a false reading.

Command Module

The command module is responsible for parsing and executing SCP (Sleek Command Protocol) commands. SCP provides the command protocol for communication between a PPA printer and its host driver. SCP is a binary language (as opposed to the ASCII formatting of the traditional PCL command language). The general command syntax is shown in Table IV. Some SCP commands are shown in Table V.

Table IV
SCP Command Structure

Command Field	Field Size	Description
ID	2 bytes	Identifies the command
Reference	2 bytes	Reference number used to cancel commands
Priority	1 byte	Order in which the command is processed
Pad	1 byte	Unused
Length	2 bytes	Number of additional data bytes
Data	0 to n bytes	Depending on command, typically contains a number of subfields

Table V
Examples of SCP Commands

Command	Description
PRINT_SWEEP	Configure hardware to print a sweep of data
HANDLE_MEDIA	Load and eject
HANDLE_PRINT_CARTRIDGE	Print cartridge change, wipe, spit, etc.
CONFIGURE_PRINT_CARTRIDGE	Print cartridge temperatures
STATUS_REQUEST/REPORT	Synchronous status information
CONFIGURE_AUTOSTATUS	Asynchronous status information
CANCEL_COMMAND/DATA	Flush a command or image data
RESTART	Reboot printer
ECHO_DATA, PERFORM_TEST, SET_ALIGNMENT_INFORMATION	Miscellaneous functions
UI_STATE, UI_MONITOR	User interface set and read
ATOMIC_COMMAND	Low-level manufacturing and test command

Command Parsing. The Parser process requests raw data bytes from the I/O module by sending a message. Command boundaries are identified, blocked, and attached to an acquired message. Each SCP command is attached to one message. This message identifies and leads the command through the system for execution. The individual messages are at first posted to the Pacer process.

Command Pacing. The Pacer process receives messages pointing to raw SCP command bytes and sorts them according to priority. It continuously selects the highest-priority command and posts that command to the appropriate module for execution (which could be I/O, mechanism, etc.) The Pacer then waits for the command to complete by waiting for a response message from the selected executor.

Commands are sent to the Pacer not only by the Parser but also by other modules that may want a command executed. For example, when the printer door is opened, a HANDLE_ PRINT_CARTRIDGE: Change_Print_Cartridge command is given to the Pacer for execution. This command is issued by the U/I module.

Command Execution. A third command module process, the Executor, executes SCP commands designated for the Parser. Generally, SCP commands are delegated to their respective modules for execution. A few commands, such as the CANCEL_COMMAND command, are executed by the command module itself.

Mechanism Module

The mechanism module executes the mechanism-related SCP commands, maintains the system's mechanical state, handles periodic print cartridge servicing needs, handles all motor needs and functions, and prints sweeps of data.

The mechanism module consists of two processes and several interrupt service routines. The top-level process, the Mechanism State Machine, manages the high-level mechanism state (cover open, paper loaded, etc.). The low-level process, the Walker/Dispatcher, manages the execution of mechanism motion scripts called ***flows***.

Mechanism State Machine. The Mechanism State Machine is a process that maintains the current mechanical state of the system, takes the proper actions when state changes occur, and returns to previous states after asynchronous state changes occur. Fig. 6 shows a small portion of the mechanism state machine to give an example of its hierarchical nature.

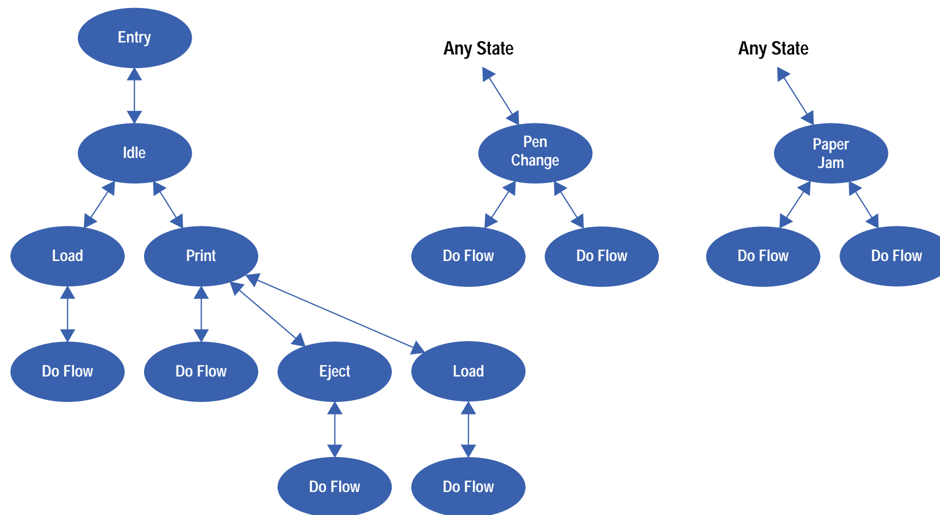

Fig. 6. *A portion of the mechanism module's hierarchical state machine.*

The mechanism starts in the Entry state and after initialization proceeds to an Idle state. As a print job comes in, a HANDLE_MEDIA: Load_Paper command causes an entry into the Load state, and when paper is loaded, to a Ready to Print state. During these states and changes, mechanism flows (or scripts) are performed and the state machine responds to asynchronous events such as a print cartridge change or paper jam. When responding to asynchronous events, an asynchronous state change is made and the appropriate flows are performed. The state then reverts back to the state that existed when the asynchronous event occurred.

Mechanism Flows. Mechanism flows are small lists of individual mechanism instructions, typically motor moves, to complete a high-level mechanical task. For instance, when starting the Load Paper state, a load paper flow is executed. This flow contains a list of individual motor move commands to accomplish the multimotor task of loading paper.

Flows are written in a custom scripting language. The reason for the custom scripting language is to permit development of motor motion without recompiling and building the firmware set. A flow can be downloaded to the printer and executed, permitting an easy standalone mechanism development environment. This technique is also used during manufacturing to invoke custom manufacturing motor movements. When a particular mechanism flow has been developed, the flow can be incorporated into the firmware set and executed just as during development. Table VI lists a few of the available flow commands.

Walker/Dispatcher. The flow script executor is a process called the Walker/Dispatcher. This process receives messages with either addresses of flows to execute or addresses of completion routines to execute. When given an address of a flow to execute, the Walker/Dispatcher looks up each opcode and calls the corresponding function to perform the opcode. When given a completion routine to execute, the Walker/Dispatcher executes the completion routine and then retries any opcode that had to wait for a completion before continuing. This is similar to a microprocessor retrying an instruction after a page fault is corrected.

Table VI
Partial List of Flow Scripting Commands

Flow Opcode	Parameters
Carriage Motor Move	Speed, position
Paper Motor Move	Speed, distance
Wait Carriage Motor Done	
Wait Paper Motor Done	
Jump to Sub Flow	Flow ID
Goto Flow	Flow ID
Fan	On/off
Relative Branch	Condition, branch distance
Exit Flow	

Actors and Gaffers. The functions that implement the flow opcodes have been nicknamed *actors*. There is one actor function for each flow opcode. A function table is used to select which actor function to execute for each flow opcode encountered.

An actor function parses the flow opcode's parameters, verifies that the particular mechanism resource isn't in use (generally a motor), and makes the appropriate call to the motor control code to start the proper motor movement. If a resource is in use preventing the actor from continuing execution, execution of the actor terminates and is retried when a resource becomes free.

A completion routine is passed to the motor control code to be executed when the motor has completed motion. These completion routines have been nicknamed *gaffers*. They deal with errors during motion, do any final cleanup, and cause the script executor to retry an actor function that couldn't execute because of a resource limitation. Gaffers aren't executed by the motor control code, but rather are posted to the Walker/Dispatcher for execution.

Motor Control. Motor control is accomplished via a combination of process and interrupt threads of execution. Generally the execution that occurs in process space would include all initial motion and interrupt setup calculations. A transition is made to the interrupt space of a selected hardware interrupt with a call to Interrupt_Context(). Once in interrupt space, calls can be made to Wait_For_Interrupt() to effectively suspend the execution until the associated interrupt occurs again. Execution continues, including any additional suspensions for additional interrupts, until time to inform the Walker/Dispatcher flow executor of completion. A message is posted to the Walker/Dispatcher with the address of the appropriate completion routine, the gaffer.

An example of a motor control function using such a combination is CM_Move_And_Hold(), which moves the carriage motor to a specific location, holds there, and posts the given completion routine. CM_Move_And_Hold() is called with a motion acceleration profile, a final position, a few other motor adjustments, and a pointer to a completion message. The function does some preprocessing to account for previous motor motion errors, to calculate the direction and distance to travel, and to select acceleration and slew parameters. The transition to interrupt space is made. The function then goes through three loops: one for accelerating, one for slewing, and one for decelerating. Each loop calls Wait_For_Interrupt() and sets up the next incremental motion request. Finally, at the completion of the motion, the function posts the completion message to the Walker/Dispatcher process.

Configuration RAM

The HP DeskJet 820C printer has a block of nonvolatile RAM that is used for configuring the printer in ways that must survive shutdowns. A C-language structure is used to organize this configuration data. Two small processes read and write the data from the nonvolatile RAM and control when this must be done. Examples of fields stored in nonvolatile RAM are shown in Table VII.

A copy of the nonvolatile RAM is kept in normal RAM. This copy is made upon startup by the Configuration process. Any process can access the configuration data copy as long as its access is protected by locking a semaphore. After a process has made any change, it must send a SAVE_CONFIGURATION message to the Configuration process. Configuration schedules the nonvolatile RAM update by sending a message to the NV RAM Process.

A second process actually reads and writes the nonvolatile RAM. This is to avoid holding up the system, since the physical reading and writing of nonvolatile RAM takes time. The NV RAM process executes at a very low priority so that nonvolatile RAM is only updated when there is nothing else to do in the system.

Table VII
Partial List of Configuration RAM Contents

Configuration Field	Description
Startup Tests	List of startup tests to perform
Print Cartridge Calibration	Stored print cartridge calibration figures
Page Count	Count of how many pages have been printed
Firm Replacements	Set of constant replacements
Alignment	Dual print cartridge alignment adjustment factors
Mechanism State	Indication of whether the mechanism was properly stored before shutdown

Power-On/Shutdown Sequencing

A process known as Pstate is used to facilitate a controlled startup and shutdown procedure. This is important to ensure that dependencies are handled during startup and shutdown. To accomplish this, a phased startup or shutdown is used. During startup phase 1, processes cannot assume that any other process has had a chance to execute any code. Each process initializes only its own internal data structures. During startup phase 2, processes can assume that all of the other processes have completed their phase 1 code. There is no hard and fast rule governing what is to be done at each phase. It is simply known that within a given phase, a process can assume that all other processes have completed all previous phases. Similar procedures are used in shutdown.

The startup sequence proceeds as follows. At startup, Pstate broadcasts to each process desiring startup information the START message. Processes indicate they want the startup information by belonging to the Startup class. Included within the START message is a phase number. The first time START is broadcast, the phase field is set to 1. Once each process has responded to this first START message, another START message is broadcast, this time with the phase field set to 2, and again, each process will respond. Finally, once all phases have been completed, Pstate broadcasts the message START_SEQUENCE_DONE. At this point, all processes can assume that the system is operational.

Internal Test Print

A small process is used to perform the internal test print feature. The Test Print process waits until it is handed the DO_TEST message. It then temporarily disables I/O and takes over the image input buffer, filling it up with test print data. To print, this process builds its own HANDLE_MEDIA: Load, PRINT_SWEEP, and HANDLE_MEDIA: Eject commands and sends them to the command Pacer for execution. Finally I/O buffers are restored and I/O reenabled.

User Interface

The user interface module, U/I, is designed to respond to stimulus of various events happening in the system. A state table is used to map a stimulus to a particular action and subsequent state. Each state also includes a set of exit conditions. The process's main function is to respond to UI_EVENT messages which are posted when front-panel changes occur.

The printer cover door and buttons generate interrupts when they change. Each of these has an interrupt service routine that takes care of debouncing, using deferred procedure calls, and posts a message to the UI process indicating the event change. The UI process then marches through its state table to make the internal change to the printer and the visual change to the user.

Printer Status

Printer status is managed by the status module. This module receives update indications from the rest of the system, composes specific status responses back to the host PC, and composes periodic autostatus responses back to the host PC.

Autostatus. Table VIII shows a sample of the autostatus data. Autostatus is a fixed structure of bits and numeric fields that is transmitted back to the host on a periodic basis. The Autostatus process is responsible for building the transmitted data block and handing it over to the I/O module for transmission to the host.

Status Update. UPDATE_ITEM messages are posted to the status module to update specific fields in the autostatus block. At this time additional notification to the rest of the system can be made by the status module. For instance, the status module will notify the U/I module of paper misloads, cover door openings, missing print cartridges, and so on.

Table VIII
Examples of Autostatus Fields

Field	Description
Misload	Paper load failed—most likely out of paper
Door Open	Cover door open
Media Jam	Paper jam detected
Print Cartridge Unaligned	Dual print cartridges not properly aligned
Last Error Code	Last error encountered by the firmware

Status Responses. The host PC can also request specific information from the printer. The Status Request process receives status request commands from the Command Pacer module, formats the result, and again hands the data over to the I/O module for transmission to the host.

The Simple Process

There are small functions or commands that must be executed that don't really fit logically into any of the modules in the system. Logically for modularity reasons, they might each form their own module or process. But in an effort to conserve ROM and RAM, these functions have been combined into a single process. Table IX shows a partial listing of the functions handled by the Simple process.

Table IX
Examples of Simple Process Functions and Commands

Function	Description
SCP Cmd NV RESET	Reinitializes nonvolatile RAM to default values
SCP Cmd SET ALIGN INFO	Stores the print cartridge alignment information received from the host
SCP Cmd SET PAGE COUNT	Stores a new value for the page counter
SCP Cmd REPLACE FIRM	Stores a new value for the specified firm
Calibration Functionality	Performs periodic calibration functions

Flash Memory Support

To provide for firmware upgrades during development and the early stages of manufacturing, flash memory is temporarily substituted for ROM. The flash memory can be reprogrammed whenever a new firmware set is available.

The SCP command language provides a command, EXECUTE DATA, which causes the firmware to jump to data downloaded into the image buffer. Before making this jump, the printer shuts down all interrupts to guarantee that none of the existing firmware is still executing. To reprogram the flash memory the downloaded program contains both the code to reprogram the flash memory and the data to program into the flash memory. When this downloaded program has completed reprogramming the flash memory, it executes a 68000 reset instruction, effectively returning control back to the flash memory and beginning execution of the newly installed firmware.

Conclusion

The HP DeskJet 820C printer firmware architecture successfully met or exceeded all cost, quality, schedule, and throughput goals. This is particularly satisfying considering the firmware platform started completely from scratch, with design leverage only in the mechanism flow scripting arena.

Acknowledgments

Special acknowledgments are in order for the HP DeskJet 820C firmware team. David Neff and Eric Ahlvin managed the team at different times. Gene Welborn started on the architecture early and had a key role in leading the concept of using message communication processes and designing the operating system interface. Mark Garboden led the design and implementation of the Walker/Dispatcher along with its associated flow scripts for motion control. John Van Boxtel designed and implemented the operating system and the low-level motor control code. Rose Elley had the critical role of providing engineering support and tools to enable R&D, customer assurance, and manufacturing to move towards the new PPA architecture. Melinda Grant did user interface design. Carl Thompsen, Bob Callaway, and Hugh Rice did significant design, consulting, and coding in the areas of print cartridge, motor, and analog functions.

▶ Go to Next Article

▶ Go to Journal Home Page