A mode change in one procedural element or equipment entity may force a change of mode in others. If an operator changes a unit procedure mode from automatic to semiautomatic, it may be wise for him or her to propagate all dependent operations and phases to semiautomatic mode. The S88 standard recognizes that propagation can move from a higher-level entity to a lower-level entity or vice versa, but it does not specify any propagation rules. Different batch management packages may have rules or may provide options for the end user.

States and Commands Associated with Batch Control

Procedural elements and equipment entities may have *states*. So can people: states of mind, a state of exhaustion, a state of undress. In regard to S88, the state is supposed to completely specify the current condition of a procedure element or equipment entity. *Commands* are one method for moving a procedural element or equipment entity from one state to another.

S88 suggests a common set of states and commands for procedural elements. Some committee members might argue that saying the standard *suggests* states and commands is too strong a statement; they'd prefer to say S88 "uses example states and commands." Regardless, the standard does not require any particular set of states and commands. However, keep in mind that the S88 committee members are pretty smart cookies. The example procedural states and commands they chose probably will work in more than 95 percent of all the batching systems used today. OpenBatch, VisualBatch, RSBatch, and Total Plant Batch all use the states and commands included in the standard. Table 8.3 is the state transition matrix for the procedural element states and commands suggested by the standard.

Table 8-3. State Transition Matrix for States and Commands Suggested by S88

		Command							
Initial State	**Next State if No Command Given**	**Start**	**Hold**	**Restart**	**Pause**	**Resume**	**Stop**	**Abort**	**Reset**
Idle		Running							
Running	Complete		Holding		Pausing		Stopping	Aborting	
Complete									Idle
Holding	Held						Stopping	Aborting	
Held				Restarting			Stopping	Aborting	
Restarting	Running		Holding				Stopping	Aborting	
Pausing	Paused		Holding				Stopping	Aborting	
Paused			Holding			Running	Stopping	Aborting	
Stopping	Stopped							Aborting	
Stopped								Aborting	Idle
Aborting	Aborted								
Aborted									Idle

Figure 8.1 shows a simplified state transition diagram for the states and commands as suggested by S88. What it doesn't show are all the commands from all the states. The diagram really only focuses on the *Idle*, *Running*, and *Complete* states. For example, if an operator issues a **HOLD** command, the batch will stop executing *Running* logic, transition to the *Holding* state, and begin executing *Holding* logic. According to Table 8.3, once the batch is in the *Holding* state, an operator can still issue **STOP** or **ABORT** commands. To keep the diagram from getting too cluttered, those commands from the *Holding* state aren't shown in Figure 8.1. (The same commands would also need to be shown from the *Held* and *Restarting* states. In addition, other lines need to be shown from the *Pausing*, *Paused*, *Stopping*, and *Stopped* states.)

Figure 8.1 State Transition Diagram for States and Commands Suggested by S88

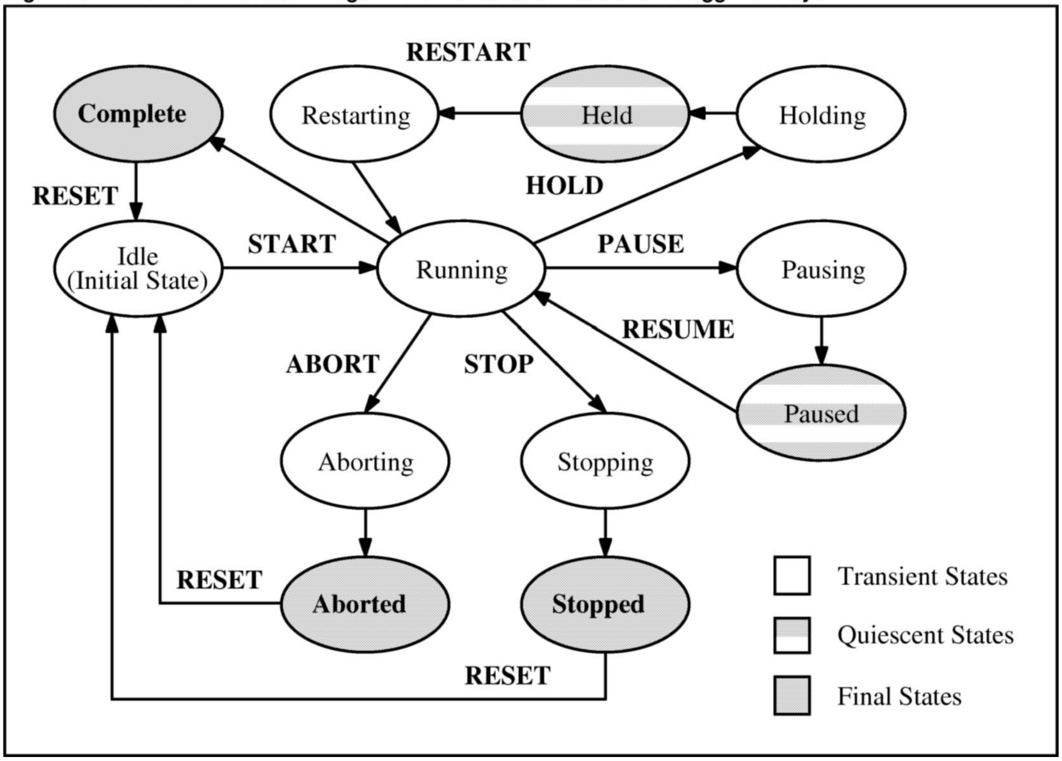


Table 8.4 describes each of the states. Table 8.5 describes each of the commands.

Table 8-4. Procedural States Suggested by S88

Idle	The procedural element is waiting for a **START** command that will cause a transition to the *Running* state.
Running	Normal operation.
Complete	Normal operation has run to a normal completion. The procedural element is now waiting for a **RESET** command that will cause a transition to the *Idle* state.
Holding	The procedural element has received a **HOLD** command and is executing its separate *Holding* logic to put the procedural element into a known condition. Once the *Holding* logic completes, the procedural element transitions automatically to the *Held* state. If no special sequencing is required to place the procedural element into a known condition, the procedural element transitions immediately to the *Held* state.
Held	The procedural element has completed its *Holding* logic and has been placed into a known or planned condition. The procedural element is now waiting for a command to proceed. This state is usually for longer-term batching interruptions.
Restarting	The procedural element has received a **RESTART** command while in the *Held* state and is executing its *Restarting* logic in order to return to the *Running* state. If no restarting sequencing is required, then the procedural element transitions immediately to the *Running* state.
Pausing	The procedural element has received a **PAUSE** command. This will cause the procedural element to stop at the next defined safe or stable location in its *Running* logic. Once the defined safe or stable location is reached, the state automatically transitions to *Paused*.
Paused	State reached once the procedural element has reached the next defined safe or stable location after a **PAUSE** command. A **RESUME** command causes a transition to the *Running* state, resuming normal operation immediately following the defined safe or stable location. This state is usually for shorter-term batching interruptions.
Stopping	The procedural element has received a **STOP** command and is executing its *Stopping* logic, which sequences a controlled, normal stop. If no stopping sequencing is required, then the procedural element transitions immediately to the *Stopped* state.
Stopped	The procedural element has completed its *Stopping* logic and is waiting for a **RESET** command to transition to the *Idle* state.
Aborting	The procedural element has received an **ABORT** command and is executing its *Aborting* logic, which sequences a quicker, but not necessarily controlled, abnormal stop. If no aborting sequencing is required, then the procedural element transitions immediately to the *Aborted* state.
Aborted	The procedural element has completed its *Aborting* logic and is waiting for a **RESET** command to transition to the *Idle* state.

Table 8-5. Procedural Commands Suggested by S88

Start	Orders the procedural element to execute its normal *Running* logic. Only valid while the procedural element is in the *Idle* state.
Hold	Orders the procedural element to execute its *Holding* logic. Only valid while the procedural element is in the *Running*, *Pausing*, *Paused*, or *Restarting* states.
Restart	Orders the procedural element to execute its *Restarting* logic to safely return to the *Running* state. Only valid while the procedural element is in the *Held* state.
Pause	Orders the procedural element to pause at the next programmed pause transition within its normal *Running* logic and await a **RESUME** command before proceeding. Only valid in the *Running* state.
Resume	Orders the procedural element to resume execution in its normal *Running* logic after it has paused at a programmed transition as a result of either a **PAUSE** command or a *Semiautomatic* mode. Only valid in the *Paused* state.
Stop	Orders the procedural element to execute its *Stopping* logic. Only valid while the procedural element is in the *Running*, *Pausing*, *Paused*, *Holding*, *Held*, or *Restarting* states.
Abort	Orders the procedural element to execute its *Aborting* logic. Valid while the procedural element is in every state *except Idle*, *Complete*, *Aborting*, and *Aborted*.
Reset	Orders the procedural element to transition to the *Idle* state. Only valid while the procedural element is in the *Complete*, *Stopped*, and *Aborted* states.

Remember that many batch control packages make the link between recipe procedures and equipment control at the phase level. The "-ing" states (*Running*, *Holding*, *Restarting*, *Stopping*, and *Aborting*) generally are controlled by separate blocks of code in a phase. For example, in a PLC each phase may be contained within a single file but have separate sections of ladder logic in that file for controlling running, holding, restarting, stopping, and aborting. We'll talk more about this in Chapters 10 and 11.

Notice that according to S88 there is a big difference between *Holding* and *Pausing*. When an operator (or procedural element) issues a **HOLD** command, a separate section of code begins executing (the *Holding* logic). When someone or something issues a **PAUSE** command, the *Running* logic simply pauses at the next appropriate location. When a **RESTART** command is issued after the batch enters the *Hold* state, still another section of code (*Restarting* logic) begins executing. When an operator issues a **RESUME** command after the batch enters the *Paused* state, the *Running* logic simply starts again at the location it paused at. You can do really neat (and really important) stuff with *Holding* and *Restarting* logic. We'll get into that in Chapters 11 and 12.

Like modes, procedural elements and equipment entities may change state when a command is given by an operator or one is generated in another procedural element. A state change can only occur when defined, required conditions for the change are met. Probably the most common way to change states is with an operator command.

Let's use a pumping control module as an example of an equipment entity changing the state. A couple of valves, a pump, and a flowmeter make up this control module. Let's say a phase issues a command to start charging an ingredient. A valve or two may open, the pump should start, and the flowmeter should soon start registering flow. If the flowmeter does not register flow within a given amount of time (maybe the pump motor tripped, a valve did not open, or a tank was empty), a failure flag will be set and a **HOLD** command automatically issued to the phase.

Also, like modes, a state change in one procedural element or equipment entity may force a change in state in others. If an operator issues a **PAUSE** command to a procedure, it may be wise for him or her to transition all dependent unit procedures, operations, and phases to the *Pausing* state. Likewise, in our flow failure example, a **HOLD** command sent to one phase might need to propagate up to the procedure level and then back down to all dependent procedural elements. Sometimes this is for the convenience of the operators so a problem can be fixed. Sometimes this is absolutely necessary for safety considerations: many processes require that ingredients charge simultaneously or at a proportional rate to each other to prevent a dangerous (or sometimes explosive) condition. Many of the batch management packages provide you with options regarding how state propagation should occur.

Looking back at Table 8.3, you can see the inherent priorities of the commands suggested by the S88 standard. The **ABORT** command has the highest priority because it can be issued from any active state (i.e., not *Idle* or *Complete*) except for the *Aborting* or *Aborted* states. Next highest in priority is **STOP**, then **HOLD**. Among the others, **RUN**, **RESTART**, **PAUSE,** and **RESUME** all have the same sort of priority because each can only be issued from one state.

States for equipment entities may be very different than states for procedural elements. For example, the states for a pump could include *on*, *off*, *percent on (or speed)*, *failed*, and *ramping*. The states for a valve could include *open*, *closed*, *percent open*, *failed*, and *traveling*. For some reason, the standard committee decided not to suggest a formal example set of equipment entity states. Maybe they were tired from a long debate on the procedural states.

Exception Handling

Wouldn't it be nice if all batches ran perfectly from start to finish? Well, we have some bad news to break to you: batches just don't behave as we would hope every time. An event that occurs outside the normal or desired behavior of making a batch is commonly called an *exception*. Handling these exceptions is an essential function of batch manufacturing and typically accounts for a large portion of the control definition. Don't be surprised if more than half of your design work and program code deals with exceptions.

Exceptions don't have to be related to failures like a tripped pump, stuck valve, or lack of steam. A tank running out of an ingredient in the middle of a batch could be considered an exception. Exception handling can occur in procedural, basic, or coordination control.

A response to an exception may cause a change in the mode or state of procedural elements or equipment entities. Using our flow failure example from the last section, a charge phase will detect an exception if the flowmeter doesn't register any flow. This causes a **HOLD** command to be issued to the phase, causing in turn a change in state from *Running* to *Holding* and eventually to *Held*.

Exception handling can be done in procedures or in equipment control. Procedure exception handling should be kept in the recipe because equipment control generally does not have any understanding of the procedures coordinating it. For example, if an operation has enough information to know that phase B and not phase A should start, handle that in the procedure, not in the phases.

You can also perform exception handling even before a recipe starts. Perhaps a recipe needs cream as an ingredient, but no cream tanks are available when the recipe starts. You can implement the exception handling in PLC or DCS phase logic. When the exception occurs, the equipment phase can issue a **HOLD** command to the batch. However, it may be better if you performed some quick checks via your HMI to see that cream is available before the batch starts.

Exception handling in equipment control is probably best handled directly in phase logic or in code that manipulates equipment or control modules. We've beat the flow failure example enough already, so consider the following example. If an ingredient can be stored in multiple tanks, you may wish to designate a primary and secondary ingredient tank. (Perhaps the operator can modify this selection.) When the phase that charges that ingredient starts, it pulls the ingredient from the primary tank. An exception occurs when the primary tank empties before the phase transfers the total amount needed. A control module monitoring the flowmeter will trigger an exception, causing the phase logic to switch to the secondary tank. This can all be done without operator intervention or holding or otherwise interrupting the batch. If the primary tank runs out and a secondary tank has not been selected, you may wish to prompt the operator for a secondary tank during the batch. Depending on your process or the volatility of your ingredients, you may not want to interrupt the batch.

Your batch management package may have recommendations regarding handling exceptions. Sequencia strongly recommends against manipulating equipment phase states directly. OpenBatch relies on an "executive" called a programmable logic interface (PLI) to monitor and maintain the states of all phases. Changing the states directly in equipment phases may confuse the poor PLI, which in turn may confuse the operators, which in turn may give you a headache.

To help manage state changes, OpenBatch allows for user-defined failures that can trigger different state changes. In the case of our flow failure example, the control module monitoring the flowmeter can set a failure flag associated with that phase. The phase logic can monitor that flag and react accordingly. At Ben & Jerry's, we created some phases that trigger a **HOLD** command if flow failures occur.

Allocating and Arbitrating Equipment Use

If batch processing equipment were really cheap, companies would purchase a whole lot of it, and no piece of equipment would be needed outside of any particular equipment train. But batch processing equipment is not cheap, so companies may have to share equipment for different batches. Sometimes equipment breaks down too, which makes it vital that you have substitute equipment on hand. Here's where *allocation* and *arbitration* come in.

As a particular batch or unit needs equipment and other resources to complete or continue processing, those resources must be assigned to it. Allocation is a form of coordination control that makes these assignments. When more than one batch or unit needs the same equipment or resource at the same time, arbitration determines who wins.

The S88 section on allocation says:

> The very nature of batch processing requires that many asynchronous activities take place in relative isolation from each other with periodic points of synchronization. Many factors, both expected and unexpected, can affect the time required by one or more of the asynchronous activities from one point of synchronization to the next. For those reasons, and because of the inherent variation in any manufacturing process, the exact equipment which will be available at the time it is needed is very difficult to predict over a significant period of time.

Wow, what a statement! A recipe does not have to be very complex to require different activities to take place separately (*asynchronously*). This is fine initially, but the resulting intermediate products must sometime later combine (*synchronously*) to produce another intermediate or a final product. When other recipes or unexpected maintenance make certain equipment unavailable, substitutes may be obtainable.

Your site may use sophisticated scheduling to try to optimize a recipe's processing sequence from the perspective of equipment usage. However, for those just-in-case situations you may wish to allow alternate equipment to be used if necessary. Recall from Chapter 4 that the path describes the usage and routing of the equipment that is necessary to make a batch. Allocation determines the path, whether fixed (the same equipment batch after batch) or dynamic (different equipment based on availability).

If more than one unit can request another resource, the resource is considered *common*. Common resources help reduce capital or operating costs while maximizing process flexibility. An expensive powder blender that serves two units is a great example of a common resource.

In S88 schemes, common resources are often implemented as equipment or control modules and may either be exclusive-use or shared-use. Only one unit at a time can use exclusive-use resources. Shared-use resources can be used by more than one unit at a time. If two units need an exclusive-use resource, one of them is going to be waiting a while.

If two units need a shared-use resource, they may both be happy. However, just because a shared-use resource is available to more than one unit does not mean it can accommodate all units at all times. For example, a glycol cooling system can be a shared-use system. However, the cooling capacity of such a system has its limitations. Two units requiring a total thermal transfer that exceeds the capacity of the glycol system might cause a slowdown in both batch processing times or, worse, an overload of the cooling system.

You must also be careful about how you design the use of these shared resources. Let's say unit A needs the glycol system and starts a pump. Then unit B needs the system. The pump is already running, but that's okay. When unit A is finished, it should not stop the pump or unit B will be affected. In this example, unit B should be responsible for shutting down the pump. This type of logic can easily be handled in equipment or control modules.

Arbitration is necessary when more than one unit or resource needs the services of another resource. This resource contention must be dealt with somehow. Here are some methods for doing so:

- Wait until the resource owner is finished with the resource. (First come-first served approach.)
- Attempt to find a substitute resource. This may include attempting to alter the path.
- Preempt the resource owner and acquire the resource based on a set of priorities. This can get complicated, especially if the resource needed contains material that is incompatible with the recipe requiring it.

All resource allocations can occur at the start of a recipe, effectively reserving resources, even though another recipe may require a reserved resource first. Allocation does not have to occur before a recipe begins. Dynamic allocation can occur as the batch is running. If more than one type of resource is available at a particular step in the process, a selection algorithm can be used to choose the best resource. Perhaps the material that makes up the resource (e.g., stainless steel versus titanium) will work with all recipes, but one material is preferred for the particular recipe running. Or one resource type processes quicker than another does. Allocation can be handled implicitly by S88 batch management software or explicitly by phase logic.

We've discussed equipment, recipes, and important control issues. Now it's time to begin exploring how to specify and design a batch management system. To do that we must first discuss what information is critical to a batch management system. That's coming up next in Chapter 9.

9

Batch Activities and Information Management (The Cactus Model)

Now that we've introduced the physical model, recipes, and how to link them, we think it's time to start talking about defining and designing a batch management system. That's the purpose of the next three chapters. Believe it or not, in our consideration of batch processing systems we've really been hanging out at a somewhat detailed level. Fasten your seatbelts, we're about to rapidly ascend to 20,000 feet. (That's about 6,096 meters for our readers using the metric system . . .)

This chapter introduces the control *functions* associated with batch manufacturing. "Wait!" you may exclaim, "haven't the last 87 or so chapters dealt with control functions?" Well, technically, no. According to S88, up to this point we've discussed control *tasks*. It appears as if the standard considers control *functions* to be a superset of control *tasks*. Additionally, control functions are grouped into control *activities*. So for review: a control activity is made up of control functions, which elaborate on control tasks.

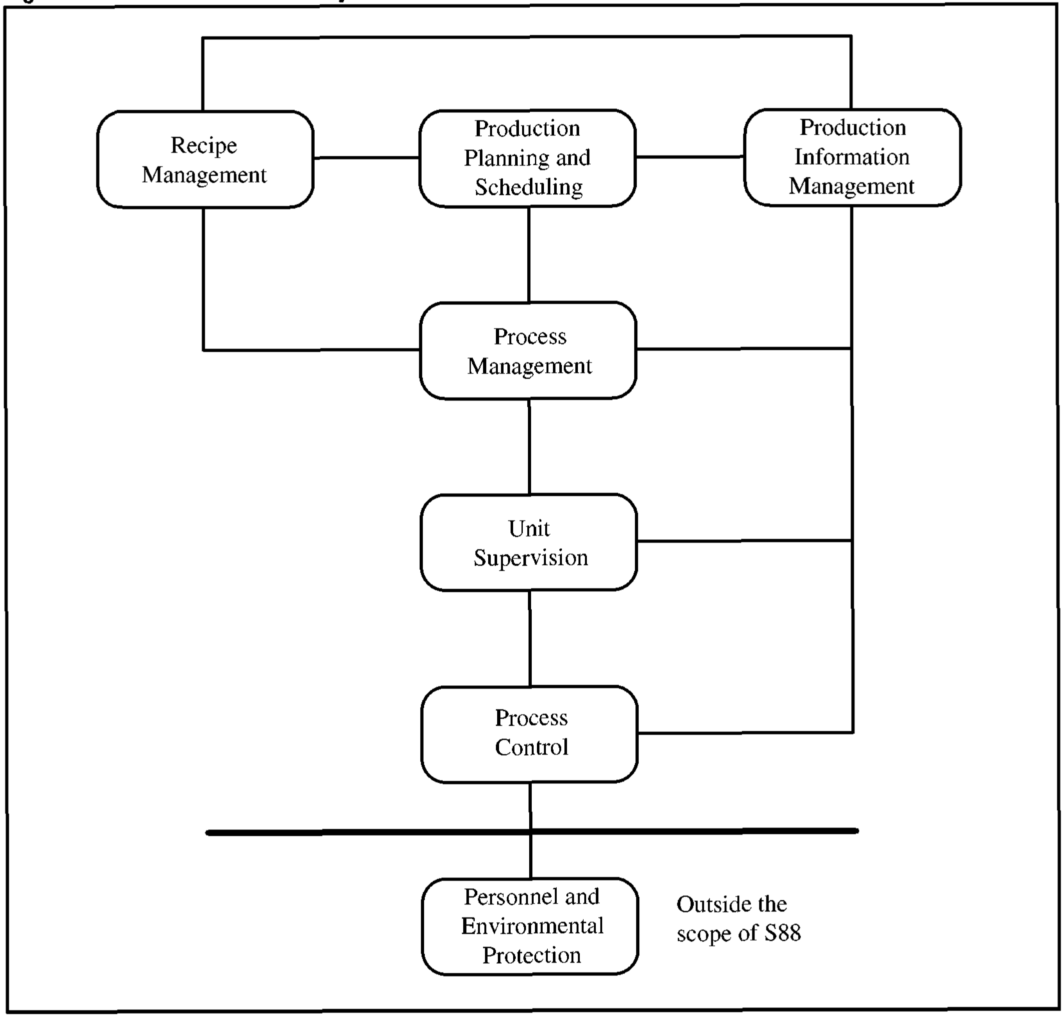
We personally believe our readers would find it much easier if we referred to these three items as *batch management activities*, *batch management functions*, and *control tasks*. However, too many S88 committee members know where we live, so we're going to stick to the terminology in the standard.

The Control Activity Model

Figure 9.1 shows the control activity model. (If you listen very closely, you may hear committee members refer to this as the *cactus model*.) While it may not seem like much now, the control activity model provides an overall perspective on batch control and shows the primary relationships between the control activities.

One important aspect of the cactus model is that it provides a description of the information that is shared between the control activities as well as between the control functions within each control activity. Depending on your perspective, the type of information needed at various parts of your process can really help drive your design. Jim has a philosophy (and Larry is tired of hearing it over and over again) that if people are a company's most important asset, then information

Figure 9.1 The Control Activity Model


Recipe Management
Production Planning and Scheduling
Production Information Management
Process Management
Unit Supervision
Process Control
Personnel and Environmental Protection
Outside the scope of S88

should be its second most important asset. However, in reality, information can be a company's number one quality problem when it's inaccurate, not timely, or not assembled correctly. If your manufacturing process is not defined well enough, you may not be collecting the right information you need to make good decisions. Get input from the right people on what information is needed. Don't guess; go to the source.

Remember that in Chapter 1 we said S88 can be applied regardless of the degree of automation? That rule also applies to elements of the cactus model. The shared information we're going to discuss in this chapter apply regardless of whether manual or automated systems are used to gather, store, analyze, and report the information. Figure 9.2 shows the cactus model with aggregate information flows that have been inserted between the control activities.