Appendix H:
GPL Advanced Concepts

Metasys advanced concepts documents are created to help continuously improve existing Metasys skills.

GPL programming satisfies life/safety applications and/or very complex HVAC systems. If the programs go beyond standard HVAC applications, programmers need to expand programming skills into full network integration. Programmers implement Priority schemes of processes and objects to ensure proper sequence of operations between standard HVAC applications and life/safety applications.

This appendix suggests ways to improve efficiency when executing in the NCM memory and help control traffic on the N1 and N2 networks.

The information provided here should increase GPL programming performances by:

• reducing programming time
• reducing traffic on the N1 and N2 busses
• reducing the time it takes to execute a GPL process in the NCM
• maximizing the effectiveness of AD and BD objects
• maximizing the use of the NCM Memory
GPL Efficiency Issues

Consider the how to address the following three GPL efficiency issues:

- a single GPL process
- all the GPL processes in an NCM
- GPL in the network

There are trade-offs to efficiency with each application having a different set of criteria. The most important criteria may be saving memory in the NCM, faster execution time, or response time of certain processes.

The NCM execution queue cannot begin until the current process is finished. Only one process is executed at a time and if a lower Priority process is executing when a Priority 1 process is triggered, the lower Priority process must complete before the Priority 1 process begins. The following sections provide suggestions for improving single GPL process efficiency.

Efficiency of a Single GPL Process

If all the attributes read by a process are triggerable, do not use a period. Any change to an attribute causes the process to execute and recalculate any changes to the control, which wastes processor time.

All period timers are supported by the operating system in the NCM. Every second the NCM’s operating system must check the period timers to see if they have expired. Although the operating system is extremely efficient, over time (years) the wasted processor time from a single unnecessary period timer can add up.

Use Binary Change-of-State (COS) Triggers Instead of Periods
If some attributes are analog and do not trigger, a period may not be necessary. If the analog value is only used when a binary trigger changes, then a period is unnecessary. If an analog value is only necessary when a binary is in a certain state (for example, room temperature only considered if in occupied mode) then the process should be programmed to conditionally set the period to a non-zero time when the binary is in a certain state (occupied mode). Conditional periods can be used to vary the period.

**Example**

A process may need to execute once a minute in occupied mode, but only once every 10 minutes in unoccupied mode. If the period is exceptionally long (12-24 hours) weekly schedule the process to run once or twice a day. Scheduling a process to execute at a time when there is less demand on the NCM can be a good alternative to a period. Weekly schedule a process that needs to execute once a day to execute when a building is unoccupied.

**Eliminate Triggers Unnecessary for Process Execution**

If a process runs periodically because of analog data, consider eliminating all or most of the triggers.

**Examples**

If a process is monitoring room temperature every 5 minutes and when room temperature is outside the comfort level, the process checks the status of certain equipment to determine if additional equipment should be turned on or off. The status attributes of the additional equipment should not trigger the process, their change of value only cause unnecessary process execution. Therefore, exempt all the triggers to the process.

If a process needs to open a damper when two fans are on and the fans are sequenced by another process such that when Fan 1 goes on Fan 2 is commanded on 10 minutes later, there is no need to trigger the process until Fan 2 is on. Therefore, exempt Fan 1 as a trigger. If Fan 1 triggered the process before Fan 2 is on, the damper would not be opened anyway, and the processor time used to execute the process would be wasted.
Set the process period for as long as possible.

**Example**

A process that needs to react to a change in outdoor air temperature probably doesn’t need to execute much faster than once every 15 minutes.

A process that only needs to run once every 15 minutes with a period of 5 minutes is executing three times as often as necessary. That makes it three times as likely to be executing when a critical process needs to execute. That means it uses up three times the NCM processor time.

For every data line out of an object block or a reference block an attribute read is done during the execution of the process.

Each attribute causes the process execution to pause. The length of the pause varies based on many factors including:

- the amount of available acquired memory
- where the data must be retrieved from (e.g., across N1, from N2 device, and traffic on either/both)
- number of messages queued up at object manager
- other activity on the NCM
- if the object does not respond within 1 minute, the process will use the last known reliable value for the attribute, flagged unreliable, and continue execution.

**Note:** If an NC’s N2 Bus is very busy, this message may return after the time-out. This will require additional processor time by the JCB-Interpreter, put an error message in the NC error log, and may affect the results of another attribute read.

There are a few rare cases where an attribute must be reread by a single execution of a process (after a command to the object or after a Wait block). For most applications, if an attribute value is needed more than once in a process, then read it once, store in a Value Holder, and fan-out of the Value Holder where needed. The Value Holder read is faster than the attribute read is to execute.
Use the Value Holder (VH) Block Instead of the Shared Variable (SVAR) Block

In general, a Value Holder block should be used when the data it contains is used only inside a single process. A shared variable is 6 bytes bigger in size and 2 bytes bigger per reference than a Value Holder. Each reference to a shared variable takes a few nanoseconds longer than a Value Holder reference. In most cases 2 bytes and a few nanoseconds are insignificant, but it is worth considering getting into good programming habits. Do not redo a GPL process in a well running NCM to use Value Holder blocks in place of shared variables.

Reduce the Number of Commands a Process Executes

For every data line out of a command block, a command is sent to an object when the line is executed. If the command block’s enable input is true or there is no enable input, a command will be sent every execution of the process. Similarly if a 2CMD block has no edge trigger and the enable in is true (or there is no enable input), then a command will be sent every execution of the process. The amount of time it takes a process to execute the command depends on several factors. However, it takes longer to create and send a command than it does to process an edge trigger on a 2CMD or a one shot pulse into the enable input of a command.

Every command sent requires the following:

• memory acquired for the command message
• object receiving the command must process it
• message may have to travel across the N1
• message may cause N2 command (although in most cases a repeated command will not cause N2 traffic)—with the SYS91 and DX91xx N2 devices, it may now be more common for the command to be repeatedly sent. When a command is received by the NC object manager, it goes out on the N2 and gets the bit state. If the bit does change then the command goes out on the N2; if it doesn’t change, there is “no” additional N2 traffic. So every command to the DX is at least one and maybe two messages on the N2.
There are a few, rare, cases where a command must be re-sent for each execution of a process even though the command does not change. In general, if a command is sent once, it does not need to be sent again until the command itself changes. When a command must be resent, determines if the process is in competition with another process, or scheduled commands, for control of the object.

If a process sends a lot of commands, consider using an MCO to issue the commands. The MCO provides automatic edge triggers.

Positioning a single GPL process in the NCM with the majority of the inputs (attribute reads) and MCOs in the NCMs with the objects to be commanded reduces N1 traffic from commands to a single command per MCO. In fire applications, sending out one command to an MCO in each NC provides faster response (damper commands) than a single process commanding objects in all the NCMs.

**Example**

For damper control, a process can perform the logic of which state the dampers should be in and send a single command to an MCO to issue the commands to all the dampers. For a fire application, an MCO in each NC would be more efficient than a single GPL process.

If a network has 5 NCs and each NC has 8 dampers to command in the event of a fire a single fire process would have to send out 40 commands to all the dampers (32 across the N1). With 5 MCOs, the fire process would only be sending 5 commands (4 across the N1), the MCOs would be commanding their slaves (dampers) in parallel and all commands would be issued quicker. The GPL process would also execute faster allowing other processes to execute in the NCM.

**Note:** Don’t use the MCO in a GPL process when there is only one or two slaves to be commanded because there may be more commands to the slave objects than if you would use standard GPL blocks and command the objects straight from GPL.
Use an MCO to trigger processes. If you want to order the execution of processes as a result of the mode change, an MCO could do that with slave order or delay times and you could create more finely tuned process prioritization.

The MCO provides edge triggering of commands because it only sends out the commands when the MCO changes state, therefore if a process sends the same command (same state) to the MCO every execution the MCO gets the command and remains at its state, if the state changes the MCO immediately starts sending out its commands for the new state.

Note: The MCO can send commands at the same time the process is commanding other objects or ending execution or another process is starting execution. This could cause many reads on the N2 or N1 LAN.

The following sections provide ways processes in an NCM can be designed to work together to make an NCM more efficient.

In GPL, for every data line out of an object block or a reference block, an attribute read is performed. Five data lines out of a single object block into five different processes means five processes reading the same value. Examine the application, if the attribute value does not need to be absolutely current, choose one of the processes (periodic for an analog value or triggered by the attribute for a binary or integer value) to be the only process that actually reads the attribute. The chosen process should write to a shared variable that all the other processes use to get the attribute value.

Reading an attribute from only one process in an NCM does all of the following:

- reduces execution time of all processes using a shared variable instead of doing read of the attribute
- reduces N2 traffic on NCM when the attribute must be read from N2 device
- reduces N1 traffic to and from NCM if object is on different NCM
- increases total NCM process memory usage by approximately 34 bytes

Efficiency of All GPL Processes in an NCM

Reduce the Number of Attribute Reads the Processes Execute

88 Appendix
If several processes read the same attributes and do the same calculation on the values, the process designated to do the reads should do the common calculations and share the results in SVARs. This reduces execution time of the other processes.

Any event which causes a process to be executed places the process on the queue (e.g., change of a triggerable non-exempt attribute or shared variable; wait, period, delay or pulse timer expiration; manual, scheduled, or programmed trigger command, etc.). The length of time the process spends on the queue is based on three things:

- process’s Priority
- processes of higher Priority in the queue
- execution time of the process currently executing

Breaking up large processes into smaller processes reduces the time a process of higher Priority waits. If a large process of Priority 3 is broken into three processes of Priority 3, any one of the smaller processes executes faster than the original. A Priority 1 process, which is queued during the execution of one of the smaller processes, waits less time to execute.

Smaller processes are easier to understand and troubleshoot. However, the trade-off is memory; each process has a fixed amount of overhead.

When breaking apart large processes, look at triggers and periods. If a process has both trigger and periods, see if the process can be broken apart so that the smaller processes have either triggers or periods. This reduces the executions of the smaller processes.

**Break Large Processes Down into Smaller Processes**

**Combine Small Processes Into Large Processes**

Note: This is the exact opposite of the efficiency described above. Larger processes are more difficult to understand, to troubleshoot, and require more acquired memory to upload and download.

Larger process are harder to debug and understand. The single large process has a longer execution time during which a higher Priority process must wait on the queue.

**Appendix** 89
Reasons to combine processes would be:

- if they were performing duplicate reads and calculations
- if the NCM has reached the 255 process limit
- If you decide to combine small processes into larger processes consider the following when doing so:
  - combine processes with the same period
  - combine processes with the same inputs

The following is a list of ways processes can be designed to improve the entire FMS Network.

**Efficiency of the Entire Network**

**Select Either AD or BD in a Process Input or Value**

Using both AD and BD in a process input or value is a waste of resources (memory and processor time). Do not define an AD or BD with an associated input and using a process to command its value (SET_BD or SET_AD) at Priority 3. Every 30 seconds an AD polls its associated input and resets its Value. Every 4 seconds (may vary on a very busy system), a BD polls its (non-triggerable) associated input attribute and resets its Value. Thus, if a process changed an AD’s value with a Priority 3 SET_AD command, the value sent by the process will be lost in less than 30 seconds. If an AD or BD is commanded at Priority 1 or 2, the value of the AD or BD will be the highest Priority commanded value. If an AD or BD with an associated input is commanded at Priority 1 or 2, the AD or BD continues to poll for its associated input.

If the Associated Inputs for BDs use triggerable attributes, then the BD will change its value when the Associated Input triggers. The Key here is if the input is triggerable. Most CS object binary attributes do not trigger (as well as other non-triggerable attributes), therefore they will be polled at the 4-second rate.

**Proper NCM Placement of a Process can Reduce N1 Traffic**

An attribute read from a process to an object on a different NCM requires two messages across the N1 (one to request the data, one to respond to the request). A process output to an object (command or write) on a different NCM requires one message across the N1. If a process inputs and outputs to multiple NCMs total up the messages in and out of each NCM and place the process in the NCM, that minimizes the N1 messages.
If a process that reads a specific attribute resides on the same NCM as an AD or BD associated to that attribute, read the value from the AD or BD instead of from the object. The reasons include:

- ADs and BDs never poll the N2 in response to a read attribute request.
- Reading an attribute from an object on the same NCM eliminates two messages across the N1.

Writing a process to replace the associated input of an AD or BD can reduce N1 and/or N2 traffic.

The following describes how an AD or BD to get its associated input.

Note: A BD with an associated input initiates this every 4 seconds. An AD with an associated input initiates this every 30 seconds.

1. A message is sent to the object for the attribute’s value:
   The requesting task (AD or BD manager) acquires memory for the message, fills in the message with the appropriate information, and sends the message to the object.

   If the object is on a different NCM, the message is sent across the N1 (via network manager). The memory acquired for the message is released. When the message arrives at the proper NCM, that NCM’s network manager acquires memory for the message.

2. Waits for a return message from the object or 1 minute, whichever occurs first.
   - If the information requires an N2 read the object manager passes the message to the N2 poll task. The N2 poll task polls the N2 device for the information, the N2 device responds to the poll.
   - The object manager (or the N2 poll task if the message was passed on) usually uses the request message to respond putting all requested attribute value information in the message. Once the response message is filled in the object manager (or the N2 poll tasks), it returns the message to the requesting task.
• If the object is on a different NCM, the message is sent back across the N1 (via network manager). The memory acquired for the message is released. When the message arrives at the requesting NCM, that NCM’s network manager acquires memory for the message. And then sends the message back to the requesting task.

3. If a message was returned it assumes the returned value as its Priority 3 value and releases the memory acquired for the message. If the message "times-out" (no return after 1 minute) it maintains the last Priority 3 value it had.

In the case of an AD, a simple process written to read the attribute from the object and command the AD to the value could greatly reduce N1 traffic if the period of the process is set to a value greater than 30 seconds.

**Example**

Assume an AD is associated to an AI whose value averages a 1% change every 10 minutes. The AD sends a message every 30 seconds requesting the AI’s value and the AI returns the message, or two request messages per minute and two returned messages per minute. A simple process that reads the AI and commands the AD to the value read, with a period of 10 minutes in place of the associated input, can greatly reduce the messages sent. The process will request the attribute value once every 10 minutes the object will return the attribute value once every 10 minutes, and the process will command the AD once every 10 minutes.

A total of three messages every 10 minutes, depending on the NCM placement of the process one, two, or three messages across the N1. Depending on the attribute that is associated with the AD, this could also reduce N2 traffic.

There are drawbacks to this approach:

• Memory. The simple process takes up considerably more memory than the associated input.

• Complexity. When viewing the AD on a Focus window, it is no longer obvious where the value is coming from extra engineering time to create the processes.

• This adds another process, which needs to be executed.
• When a process reads an unreliable value, it uses the last known reliable value for the attribute, flagged unreliable. So, the AD would be commanded to the last known reliable value of the attribute, not the attribute's unreliable value. The use of the AD should be considered when contemplating this approach.

Reducing N1 traffic can increase the efficiency of other NCMs as well. Consider how many messages the NCM with outdoor air temperature must handle in a system with many NCMs all needing outdoor air temperature. Every N1 message eliminated is one less message that needs to be acquired.

4. Writing a process to replace the associated input of multiple ADs or BDs can reduce N1 and/or N2 traffic. Writing a single process that reads an attribute and sends (set commands) values to multiple ADs or BDs can significantly reduce N1 and N2 traffic. See the earlier outdoor air temperature example.

   The drawbacks to this approach are the same as Step 3 above.

   Using an AD or BD in the same NC could eliminate GPL code. An AD or BD can be assigned an initial value; the initial value could eliminate special code written to determine if an NC is responding. If the GPL process does an attribute read and never gets a response, it uses 0 or false as the default. Using an Initial value can provide a better value to use in case NC to NC communication is unavailable.
## GPL Libraries

### HLIB - GPL HVAC Library

If you need information about a compound in the GPL HVAC library refer to the appropriate application notes in the *Metasys Network Technical Manual (FAN 636)*.

The same information in the text of the application notes is available through the Description icon. Highlight the compound name and click on the Description icon. Page Up and Page Down keys move you through the text.

The text for the example files is available in the same way.

### Recommendations

We recommend when you modify or add any compound or example files to the GPL HVAC library, you also modify or add the corresponding descriptor file. It may be helpful to make additions to the GPL HVAC library or create a new library to create a standard for applications that are unique.

### Using the Metasys GPL HVAC Library

The easiest way to build a GPL library is that every time you make a compound, save it to the disk instead of the screen. Then load the compound into your strategy file when you need it.

### Building Your Own Library of GPL Compounds

When you make a process, keep the template a Group compound. Give it a name that helps describe the compound and save it to the disk. Then when you need it, load it into your strategy file and change it to a process compound.

Don’t forget to share your library compounds with others in the company.
In order to understand process prioritization, some background information on the process queues is needed. There are five process queues in the NCM, one queue for each of the four priorities and a fifth for TIME SLICED processes (see the Process Statuses section for an explanation of Time Sliced processes). A queue is nothing more than list. In this case the lists are of processes requesting execution. With two exceptions, whenever an event occurs that requires a process to be executed the process is placed at the bottom of its Priority queue. The two exceptions are:

- when the process is already on its Priority queue or Time Sliced.
- when the process is in a state (Disabled, Error) that cannot be executed.

If a trigger is received for a process on its Priority queue in a Wait state the process maintains its position on the queue, however, its Wait state is canceled.

Processes are removed from the queues one process at a time and executed.

The process that has been on the Priority 1 queue the longest (i.e., the one on the top of the list) executes next. If there is no process on the Priority 1 queue then the one that has been on the Priority 2 queue the longest executes. If there is no process on the Priority 2 queue then the one that has been on the Priority 3 queue the longest executes and the same is true for Priority 4. The last to execute is any process that has been Time Sliced.

The order in which processes are removed from the queues explains the effects of process prioritization. In a NCM with processes of all priorities, a Priority 2 process spends less time on the queue between the event that caused it to be queued and its execution than an identical process with a Priority of 3. Process Priority should be assigned based on the application. The most critical and ideally the least frequent processes should have the highest Priority.

Process Priority cannot guarantee that a critical process will execute immediately or at a constant period. When any event causes a process to be queued the only way it will be executed immediately is if there is no process currently executing. Only one process at a time is executed.
Each process must finish execution before another process can begin execution. Once a process has started, even if it is at Priority 4, it will continue until it finishes. This is true even if a Priority 1 process was added to the queue immediately after the Priority 4 started. Finish execution means all blocks executed, or a STOP, ABRT, or Wait block is executed, or the process is ERROR or Time Sliced, or the process is commanded DISABLED. Therefore, if there is a process that absolutely, positively must execute immediately, then it absolutely, positively must be the only ENABLED process (other than the RESTART process) in the NCM.

There are two types of memory in the NCM: allocated and acquired. Allocated memory is used for long-term or permanent storage. This includes the NCM programs themselves, processes, databases, etc. Acquired memory on the other hand is used for short term storage such as sending messages between tasks in the NCM, sending or receiving messages on the N1 or N2, sending commands to objects, etc. During process execution a process may acquire memory. When a process acquires memory it is for the purpose of communication. The communication is to obtain information or send information. The communication may be to objects and features in the same NCM or a different NCM, it may be to a printer or OWS. The following are all the times a process will acquire memory:

- to read an attribute value (data line out of object block or REF block)
- to read a totalization value (TOT block)
- to time a period or wait (Wait blocks, STOP blocks or all blocks executed)
- to send an advisory message (ADV block)
- to send a print message (PRNT block)
- to time a pulse or delay (PULS, DLAY, and BSEQ blocks)
- to send a command (CMD or 2CMD blocks)
- to write an attribute value (WRIT block)
- to trigger for a binary shared variable change (trigger other processes)
When a process acquires memory, it affects the NCM that the process is executing in, but it may also affect other NCMs on the network. The following may cause memory to be acquired in another NCM when the object read/commanded or the printer written to is associated with another NCM:

- to read an attribute value
- to read a totalization value
- to send an advisory message
- to send a print message
- to send a command
- to write an attribute value (WRIT block)

A process has only one of the following acquired at a time:

- to read an attribute value
- to read a totalization value
- to time a period or wait

A process acquires the following and may have many of these acquired at a time:

- to send an advisory message
- to send a print message
- to time a pulse or delay (PULS, DLAY and BSEQ blocks)
- to send a command (CMD and 2CMD blocks)
- to write an attribute value (WRIT block)
- to trigger for a binary shared variable change (this does not affect other NCMs)

The difference between the two categories listed above is the one at a time category issues a message and does not proceed until the message is returned. The many at a time category issues a message and continues with the potential to immediately issue another.
Example

A one at a time read attribute value sends the message and pauses execution until the message is returned; during the pauses, it is not executing, it is not asking for any other messages. An example of the many at a time is a command sent to an object with process execution continuing, the process may immediately send another command. Several commands may be sent by a single process before the first commanded object has processed the command.

Note: An NCM has a single processor and each piece (object/feature) must share the use of the processor, thus an executing process may send many messages before an object gets its turn.

If an NCM is running out of (or running low on) acquired memory, the following blocks in a GPL process may be contributing to the problem:

Note: Processes are not the only users of acquired memory, however, this document deals strictly with processes.

- ADV (advisory) block
- PRNT (print) block
- PULS (pulse) block (except one-shot)
- DLAY (delay) block (except one-shot)
- BSEQ (binary sequencer) block
- CMD block
- 2CMD block
- WRIT block
- binary SVAR blocks with any non-exempted output connections

Advisory and print messages are particularly large and a printer is a slow device. A process was not designed to be a report generator. Use print messages with extreme care. The timer messages (period, wait, pulse, delay, and binary sequencer) stick around for the length of the timer, or until they are canceled. The other messages only exist until they are processed by their receiver. In the case if Print Messages, they may accumulate in the NCM that has the printer.
Shared Variables

A shared variable is a data storage location in a NCM. A shared variable’s purpose is to give a name to a specific data location so the data in the data storage location can be shared between processes on the same NCM. In GPL there are two ways to create this data storage location:

- a SVAR block
- by drawing a data connection line that connects two operation blocks across two different processes.

In JC-BASIC this data storage location is created by declaring a variable SHARED.

In GPL or in JC-BASIC a shared variable should be used when data (usually calculated) in one process needs to be used by another process.

A binary shared variable that is not exempted in a process that reads its value triggers the process each time the value changes. Therefore, when using a binary SVAR block to reduce attribute reads exempt the shared variable in the process that reads the attribute and assigns the attribute's value to the SVAR.

Using Shared Variables

The following is a list of ways a shared variable can be used to increase process and NCM efficiency. If the data does not need to be shared with another process, use a Value Holder block instead of the Shared Variable block for greatest efficiency. In JC-BASIC, if the data does not need to be shared with another process use a local variable.

1. Instead of doing a fan-out from an object or REF block, drag a single line out to an SVAR block and do the fan-out from the SVAR block to reduce N1 traffic and/or N2 traffic.

   This does a single read of the attribute value, at most one poll of the N2, and at most one read and reply on the N1. By using the SVAR block the attribute value does not change during the process, every comparison uses the same value of the attribute.
Example

If a process wants to start a fan with the temperature between 60 and 80 degrees, there are two compare blocks both needing the temperature value. If a shared variable is used to store the temperature, both compare blocks will get the identical temperature value; if both compares have a line direct to the object block, two reads of the object (probably to the N2 device) will occur, and chances are the value may vary slightly between reads. In this example, a few tenths or hundredths of a degree difference may not matter; however, you may have an application where it does matter. Also, if the attribute value is binary, the change in value could make a dramatic difference. The first half of a process could execute as if the BI were open and the second half could execute as if the BI were closed.

2. A shared variable takes up less NCM memory than an AD or BD object. Therefore, if a calculated value does not need to be seen by the customer or used by another NCM’s process but does need to be used by multiple processes on a single NCM, put it in a shared variable to save memory. The shared variable will have an added bonus; in this case, it is faster to access a shared variable than to access an AD or BD attribute.

3. Use a shared variable as described above to decrease process execution time and reduce N1/N2 traffic. Each attribute read involves a pause of up to 1 minute while message passing occurs to retrieve the data. Reducing the number of reads (even on the same NCM) decreases process execution time.

If the data does not need to be shared with another process, use a Value Holder block instead of the Shared Variable block for greatest efficiency. In JC-BASIC if the data does not need to be shared with another process use a local variable.
Some facts that may help you make wise choices concerning when to use a Value Holder block (local variable in the NCM) and when to use a Shared Variable block (shared variable in the NCM) follow:

1. In the NCM, a local variable and a shared variable do not use the same amount of memory to store the variable data (name, value, reliability, array information). However, a shared variable must store extra information to provide its extra capabilities. Each shared variable has 6 bytes more of information per variable than a local variable (2 bytes for location information, 4 bytes that signify whether the variable is a trigger to any process).

If the shared variable is binary and triggers any process, it uses an additional 32 bytes (regardless of how many processes it triggers) to store the information on which processes to trigger. The shared variable has a small amount of overhead that a local variable does not, the overhead averages 1/4 byte per shared variable.

2. Each NCM process object contains references to shared and local variables. The reference to a shared variable is a long pointer, the reference to a local variable is a near pointer. The ramifications this has to you is a long pointer is 4 bytes, a near pointer is 2 bytes. A long pointer takes a few nanoseconds longer to execute than a short pointer. In most cases, 2 bytes and a few nanoseconds are insignificant, but it is worth considering getting into good programming habits. There is a variable reference in the NCM process object every time the variable value is read or written, in GPL this is every line in or out of a SVAR or VH block. In JC-BASIC, this is every time a local or shared variable name is used.
3. If NCM memory is extremely full shared variables can be used instead of local variables to save memory. Instead of having a VH block in every process, reusing a SVAR could save memory. The memory saved is the memory each local variable would take up vs. the overhead of a shared variable. There are some important considerations:

   a. Space **must** be more important than speed.

   b. Space **must** be more important than program readability.

   c. The VH block **must** be assigned a new value before it is used elsewhere for every process substituting the common SVAR. Another way of saying this is that the value assigned to the Value Holder now will not be used for the next execution of the process, the value is reassigned (either by a calculation or attribute read).

   d. There are enough processes in the NCM to justify this substitution (if the type is binary all processes must exempt it as a trigger) the VH block has less than 12 connections (12 or more connections no longer saves memory because the process object code required for an access of a shared variable is 2 bytes per access larger than the process object code for VH’s local variable) **extreme caution** should be used if making this substitution, and this is not recommended as a good programming practice. Other options should be explored before attempting this.

4. A shared variable can be used to decrease N1/N2 traffic and increase an NCM’s performance as follows:

   - Analyze all processes, which read the same attribute.

   - Determine the acceptable age of the attribute’s value that will provide proper operation. Acceptable age means how old can the value be, for example, since outdoor air usually doesn’t change very fast, an acceptable age may be 15-20 minutes.
• Choose a process accessing the attribute value whose period is less than or equal to the acceptable age. For the outdoor air example, if three processes use the variable, with periods of 50 seconds, 2 minutes and 5 minutes, then choose the 5 minute process since its period falls closest to the acceptable age.

• In the chosen process, read the attribute value and store in a shared variable in the chosen process and all other processes. In the outdoor air example (assuming outdoor air on another NCM), this reduces N1 traffic to read outdoor air from an average of 3.4 messages per minute to an average of 1 message per 2.5 minutes (a read is 2 messages, a request and a reply). In the outdoor air example (regardless of the NCM containing outdoor air) this reduces N2 traffic generated by these processes from 1.7 polls per minute to 0.4 polls per minute for outdoor air. In addition all processes not reading the attribute will reduce their execution time by the time that they previously spent creating the message, waiting for the response, and cleaning up after the message.

---

**Process Statuses**

The process compound object in the archive database is referred to in this document as a disk process. The process compound object in the online database of the NCM is referred to in this document as a Field process. The terms Disk and Field are taken from the compound’s System name process window.

A disk process has two states: enabled and disabled. A process in the disabled state (disk) is in the disabled state (field) when download is complete. A process in the enabled state (disk) downloads into the Ready state (field). A process in a single process downloads into the Held state.

A process has eight statuses that can appear in the field process focus window:

1. **Not fully downloaded**

   This status would appear in the focus window while a process is being downloaded. It will also appear when an error occurred during download and the process cannot be executed because of the error.
2. Error

A process can be in the Error state for two reasons:

- the process executed an abort (ABRT block in GPL or an ABORT statement in JC-BASIC)
- an unrecoverable error, usually floating point, occurred. Whenever an unrecoverable error places the process in the Error state an advisory is generated by the process.

A process in the Error state does not execute, as long as the process is in the Error state. No trigger will cause a process in the Error state to execute. A process can be changed out of the Error state, by an enable command (manual or programmed), or by downloading a process of the same name (system\object name).

3. Disabled

A process can be in the Disabled state for two reasons:

- the process received a disable command (manual or programmed)
- the process was downloaded in the disabled state

A process in the Disabled state will not execute, as long as the process is in the Disabled state. No trigger will cause a process in the Disabled state to execute. A process can be changed out of the Disabled state by an enable command (manual or programmed), or by downloading a process of the same name in the enabled state (disk).

4. Held

A process can be held for four reasons:

- A process is placed in the Held state when a full download of the NCM has completed, the process was downloaded in the enabled state (disk), and a restart process was downloaded.
- A process is placed in the Held state when the NCM it resides in resets, the process was in the Ready, Executing, Waiting, Time Sliced, or Held state when the NCM reset and a restart process exists in the NCM.
• A restart process cannot be placed in the Held state. If the NCM does not have a Restart process, no processes in the NCM can be in the Held state.

• When a restart process exists in the NCM, no other process will execute until the restart process has successfully completed. The restart process completes successfully when it has executed a stop (STOP block in GPL or stop statement in JC-BASIC) or has reached the end of the process (all blocks executed in GPL, end process statement in JC-BASIC). If the restart process is in the disabled or not fully downloaded state when a full download completes or if the restart process is in the disabled, error, or not fully downloaded state when the NCM resets no other process executes. All processes placed in the Held state remain in the Held state. If the restart process finishes unsuccessfully and is placed in the Error state (see above), all processes placed in the Held state remain in the Held state.

5. Ready

A process in the Ready state is available for execution when triggered. The only process in the NCM in the Ready state that cannot be triggered is the restart process. A process is placed in the Ready state when:

• the process finishes execution by executing a stop (GPL STOP block or JC-BASIC stop statement) or by executing a end process (all GPL blocks executed or JC-BASIC end process)

• the process is disabled and receives an enable command (manual or programmed)

• the process is downloaded (single process download) successfully in the enabled state (disk). The restart process has finished if a restart process exists on the NCM.

• the process is in the Waiting state and the process is triggered
6. **Executing**

The process is in the Executing state when the process is being executed. The Interpreter is the task in the NCM that executes a process. When the process is removed from the queue for execution the interpreter changes the status to executing. A process must be in the Ready, Waiting, or Time Sliced state to be on the queue. When the process is done executing it is placed in the Ready or Waiting state if the process executed normally. The process is placed in the Error state if the process executed abnormally. The process is placed in the Time Sliced state if the process is in an infinite loop or too large (see below).

The process is placed in the Disabled state if a disable command (manual or programmed) is received for the process, in this case where a process is executing, when a disable is received the process, halts wherever it is, in the code. Care should be taken when disabling a process that does two or more commands that if the first command occurs the second command must be sent. For example, if the pump is started the damper must be commanded open, consider that if the disabled command is received after the pump is started but before the camper is commanded the damper command will not be issued.

Only one process per NCM can be in the Executing state at a time.

7. **Waiting**

The process is placed in the Waiting state when the process executes a wait (GPL Wait block or JC-BASIC wait statement). When the wait timer expires, the process is put on its Priority queue. The process remains in the Waiting state until:

- the wait timer has expired and the process is removed from the queue for execution, at which time the process state is changed to Executing.
- a trigger is received for the process. The process state is changed to Ready. If the wait timer has not expired, the wait timer is canceled and the process is placed on its Priority queue. If the wait timer has expired, the process stays on its Priority queue where it had been placed by the wait timer expiration.
Any of the following can trigger the process:

- non-exempt triggerable attribute read by the process
- manual trigger command
- programmed trigger command
- scheduled trigger command
- binary shared variable read by the process
- expiration of the pulse timer (BSEQ, PULS, and DLAY blocks or JC-BASIC canc_pulse and n_canc_pulse functions)

The following removes the process from the Wait state:

- the process receives a disable command (manual or triggered) and the process is changed to the Disabled state. If the wait timer has not expired, the timer is canceled, or if the wait timer has expired, the process is removed from the queue.

8. Time Sliced

Each time a process’s status is set to Executing, an op-code counter is set to 32767. For every op-code executed in the process, the counter is decremented. If the counter reaches zero, the process is Time Sliced.

When a process is GPL translated, it is converted to JC-BASIC source code. When a process is compiled, it is converted from JC-BASIC source code to process op-codes that the NCM's Interpreter task can execute. The number of op-codes compiled per GPL block or JC-BASIC statement varies greatly. The number 32768 is large enough that any reasonable application can execute without being Time Sliced. The maximum size of a process is 32 Kb; only a portion of the 32 Kb can be op-codes. If a process exceeds its op-code count there is a loop where it is executing some of the op-codes more than once. If a process executes more than 32768 op-codes, it may be in an infinite loop.

Once a process exceeds it 32768 op-codes its state is changed to Time Sliced and it is placed on the Time Sliced queue and an advisory message is issued. A process is removed from the Time Sliced queue and execution continues at the exact op-code it left off whenever there are no processes on the Priority queues.
Each time execution of a Time Sliced process is begun, the op-code counter is divided by 2 until the op-code counter reaches 500 or less (i.e., if the process is removed from the Time sliced queue for the first time it will start with a op-code counter of 16383, if the process is removed from the Time sliced queue for the fifth consecutive time the op-code counter will start at 1027). A process with a status of Time Sliced cannot be triggered.

When a Time Sliced process has been removed from the queue for execution its status is changed to Executing, except for the reduced op-code counter it is executed the same as any other process.

A disable command (manual or programmed) will change a Time Sliced process to the disabled state.

Process status is a readable attribute of a process object. The process status attribute is an integer attribute. The status attribute has a range of 1-8. Listed below is the integer value of the process status and the corresponding state the process is in:

<table>
<thead>
<tr>
<th>Attribute Value</th>
<th>Process Status (field)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Not Fully Downloaded</td>
</tr>
<tr>
<td>2</td>
<td>Error</td>
</tr>
<tr>
<td>3</td>
<td>Disabled</td>
</tr>
<tr>
<td>4</td>
<td>Held</td>
</tr>
<tr>
<td>5</td>
<td>Ready</td>
</tr>
<tr>
<td>6</td>
<td>Executing</td>
</tr>
<tr>
<td>7</td>
<td>Waiting</td>
</tr>
<tr>
<td>8</td>
<td>Time Sliced</td>
</tr>
</tbody>
</table>

JC-BASIC allows reading any readable integer attribute. A GPL user’s block can be used to access the status attribute of any field process. The attribute name is STATUS.
**When to Use the Restart Process**

Some processes may go unreliable the first time they run after the NCM starts up because some of the field objects being read by the processes are initially offline/unreliable. This happens when the field objects have not reported to the NCM at the time the processes run, but eventually come online.

If you have this situation, then create a Restart process that includes a Wait block with a wait time of xx seconds. If a Restart process already exists, add the Wait block to it. This Wait block should be placed in the process so it executes before any objects are read. The amount of wait time needed will be job dependent but you may wish to start at 30 seconds and decrease the value if no problems occur.

**How a Process Period Works**

The process period timer is set when the execution of the process is complete and the period is not 00:00:00. The process period timer is not set if the process goes into a Wait state (a Wait block in GPL or a Wait statement in JC-BASIC). The process period timer is not set if the process is put into the Time Sliced or Error states. When the process period timer expires the process is placed on the queue and will be executed as soon as possible based on process Priority.

A fixed process period does not guarantee that the process will execute at that rate. For example, a process with a 5-minute period and no triggers will execute every 5+x minutes where x is variable and is the sum of the process execution time, the time spent on the queue, and any time spent Waiting. The process execution time varies based on the size and complexity of the process object code, the response time of attribute reads, and the other NCM activity while the process is executing. The time spent on the queue is based on the Priority of the process and the execution time of all processes ahead of it on the queue (either higher Priority or same Priority but queued first).

A process period timer is canceled whenever the process is triggered. Therefore, a process with a 5 minute period timer that has been timing for 2.5 minutes will be placed on the queue immediately if any trigger for the process is received. A process period need not be fixed. The period can be calculated. If a process has more than one Period statement (block) the last one executed determines the length of the process’s period to be started at the end of the current execution.
A process period of 00:00:00 does not start the period timer at the end of execution, therefore a process with a period of 00:00:00 will not execute again until it is triggered.

When set, the process period timer is set to the value last assigned to the period (process template’s period or PERD block in GPL, PERIOD statement in JC-BASIC). The timer has a range of 1 second to 23 hours-59 minutes-59 seconds. When the period timer expires, the process is placed on a queue and executes in order based on process Priority and time on the queue.

The process period timer is canceled and the process is immediately queued if any event occurs that requires a process to be executed (change of a triggerable, non-exempted attribute or shared variable; delay or pulse timer expires; manual, scheduled, or programmed trigger command). The process period timer is canceled and the process is not queued for execution if the process is Disabled.
**Frequently Asked Questions**

**Question**
If a process is executing and encounters a wait or delay block, say 10 minutes, will other processes waiting to be executed go ahead and run during the 10-minute delay or will they set until the first process has fully executed?

**Answer**
The wait and delay blocks have different actions. The Wait block when executed immediately stops the current process execution (as long as wait is not 00:00:00), sets the process status to Waiting and lets the other processes on the queue execute. A Wait block stops process execution immediately, means that all blocks whose execution follows the Wait block in the process will not execute until the Wait timer expires.

When the wait timer expires, execution of the process begins at the block immediately following the WAIT block. When a delay or pulse block timer expires, the execution of the process starts at the beginning of the process.

The delay and pulse blocks do not stop the execution of the process. When they set their timer, the process continues to execute the process. A process can have multiple delay and pulse timers active at the same time (one for each delay and pulse block). Expiration of a delay timer does not cancel the timer of other delay or pulse timers (expiration of a delay or pulse timer will cancel the WAIT timer).

Only one process can be executed at a time in the NCM. The process that is executing has a process status of EXECUTING. Whenever the Interpreter changes the process from EXECUTING to any other status, the Interpreter is immediately available to execute another process.

**Question**
Will a CS object (Reference Block) trigger a GPL process, or must you use a process timer?

**Answer**
Some attributes of a CS object are triggerable. (See the CS object attribute list in Appendix G: Attributes for triggerable attributes). When using the other CS object attributes, a process timer is necessary. If in doubt about the triggers of a GPL process, check the trigger table in the compile listing.
Question: Is last known value table cleared when NCM reboots?

Answer: First, let’s redefine reboot to be coldstart and warmstart. A coldstart is hitting the reset button or performing a full download, and warmstart is after a powerfail where the battery has maintained memory.

On a coldstart, the last known value table is built by the process object manager as processes are downloaded. As the last known value table is built the last known values are set to 0.0 for real, 0 for integer, 00:00:00 for time, and False for Boolean.

On a warmstart, the last known value table is maintained. The last known values are not changed.

Likewise, shared variables are initialized on a coldstart as they are downloaded, and shared variable values are maintained on a warmstart. In contrast, Value Holders and other local variables are initialized on both a coldstart and on a warmstart.

Question: What is the maximum number of commands an NCM can execute per minute?

Answer: There are 600 Commands per minute.

These commands come from the GPL processes executing as well as features that produce reads and writes to objects.

For the DX, every command requires three messages on the N2. For every DX command, the bits are read (one message to the DX to read the bits, one message back with the data); if the command changes the bits then a third message with the new settings is sent on the N2.