Job Waits and iDoctor for IBM i White Paper

(for IBM i 6.1/7.1/7.2)

Version 5.0 – May 11th, 2015

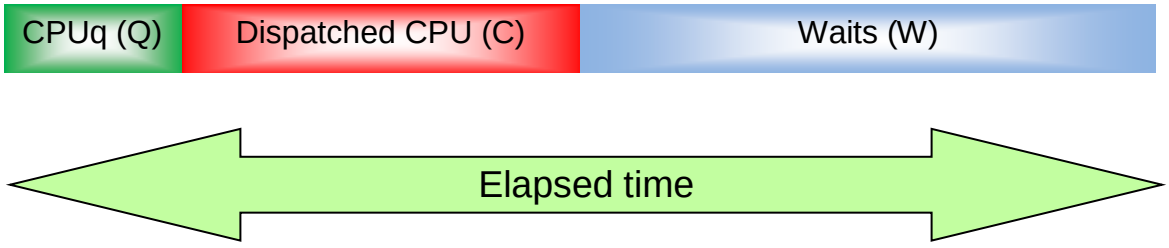
Table of Contents

1.0 Running and Waiting

All "units of work" [1] in a system at any instant in time are in one of three states:

Q. **CPU Queuing**: ready to use CPU, but waiting for a processor to become available (a.k.a. "ready", "CPU queued", "CPUq")
C. **Dispatched CPU:** (a.k.a. "dispatched to a CPU/processor", "on a CPU", "running", "running and sharing a processor"). ***Note****: Dispatched CPU time very frequently differs significantly from CPU utilization or CPU time due to multithreading (hardware or software), virtual processors, or background assisting tasks.* ***This is normal!*** *More on that later.*
W. **Waiting** for something or someone (a.k.a. "blocked", "idle")

A thread's *Run/Wait Time Signature* might look like:



How much time a unit of work spends in state Q is a function of the amount of CPU competition it experiences while getting its work done. However, note that the CPU queuing described here reflects the amount of time spent waiting to become dispatched to a processor.

How much time a unit of work spends in state C depends on the program design and how much work it's requested to perform. Factors such as hardware/software multithreading settings, and the amount of CPU that's been assigned to the partition will affect how much dispatched CPU time is spent sharing the processor with other threads.

[1] A "unit of work" is a single threaded job, each thread in a multi-threaded job or a system task.

How much time a unit of work spends in state W depends on many factors. But at this point we need to differentiate two types of waits:

A. Waiting for a work request to arrive (a.k.a. idle)
B. Waits that occur while performing a work request (a.k.a. blocked)

Type A waits, for example, in interactive work would be considered "key/think time". These waits are typically not a "problem". Or if they ARE a problem, it's usually one external to the machine they are observed on (e.g. a communications problem causing slow arrival of work requests). Note: batch work rarely has any type A waits, unless the batch work is driven, for example, by a data queue… and the data queue is empty.

Type B waits are the interesting ones. While it's debatable whether or not all these types of waits should be considered "problems", the following is a safe and valid statement:

"Outside of CPU usage and contention, type B waits are the reason jobs/threads take as long as they to do complete their work."

So, a more refined Run/Wait Signature for an interactive job/thread might look like:

And a typical batch type job/thread would look like:

1.1 Level set

This discussion applies to individual units of work… single threaded jobs and individual job threads. Many modern application engines involve the use of more than one job and/or more than one thread to process each "transaction". The ideas presented in this document still apply in those cases, but each unit of work must be individually analyzed. There's an additional burden placed on the analysis process to tie together the flow of work across the multiple jobs/threads. And such modern transaction engines frequently make it difficult to differentiate between type A and type B waits.

1.2 The Mysteries of Waiting

The waiting component of a job/thread's life is easy to compute, but rarely discussed and scrutinized.

For batch type work:

```
Waits = Elapsed Time – CPU Time[2]
```

For interactive type work:

```
Waits = Elapsed Time – CPU Time – Key/Think Time[3]
```

What is the reason why waits have historically been ignored, unless they become so severe that the elapsed time difference becomes painfully obvious? Suggested answer: because little instrumentation or tools exist to measure and provide detail on waits. Waits are the "slightly off" relative that lives in the basement. Unless his demand for food becomes excessive, or the music gets too loud, he is best ignored. You certainly don't want to talk about him with friends.

1.3 Are waits "bad"?

This paper contends the answer is "yes". (We are obviously talking about type B waits). There's a common misconception that a job/thread that "uses high CPU" is intrinsically bad. It MIGHT be bad. For example: If a work process normally takes 2 hours to complete with 45 minutes of CPU and, after a software or data change, now takes 4 hours with 3 hours of CPU, that IS bad. But just looking at a job/thread (in a non-comparative way) that uses a high percentage of CPU, and declaring it "bad" misses the point that "the lack or minimal occurrences of type B waits is a GOOD thing". For batch type work (that does not have type A waits, where it is waiting for work to arrive), if the type B waits are reduced/eliminated, the job/thread's "CPU Density" [4] increases. Ultimately, it could use 100% of a processor[5].

Let's take an example: A batch job that runs for 6 hours and uses 117 minutes of CPU. The first thing to consider is how much time of the "wasted" 243 minutes of elapsed time was CPU queuing (i.e. contending/waiting for a processor). This paper will go on to demonstrate how this value, and all the waits, can be measured in great detail. But for this example, let's suppose that 71 minutes of CPU queuing was involved. This means that the job was in type B waits 172 minutes. This means that the job could potentially run in 3 hours and 8 minutes… if the type B waits were completely eliminated. Contrast this with how the job might perform if the CPU speeds on the machine were doubled. One

---

[2] Assumes CPU Queuing is not significant

[3] Assumes CPU Queuing is not significant

[4] If a single thread consumes all of a single processor for a period of time, it is 100% CPU dense. If it consumes 1/8th of a process for the same period, it is 12.5% CPU dense. This is true regardless of the number of processors on the system or in the partition. For systems with more than one CPU in the partition, CPU density is NOT what is seen on WRKACTJOB or WRKSYSACT commands. But can be computed from those, knowing how many CPUs are available to the job.

[5] DB2 Multitasking can make a job/thread appear to use more than 100% of a processor, as the background assisting tasks promote their CPU consumption numbers into the client job/thread. Note: this can also make accurate capacity planning more difficult.

would expect the CPU minutes and CPU queuing minutes to be halved, yielding a job run time of 4.5 hours. Summary: eliminating the type B waits could have the job run in 3 hours 8 minutes. Doubling the CPU capacity could have the job run in 4 hours, 30 minutes**. Conclusion: wait analysis and reduction can be a very powerful, cost-effective way of improving response time and throughput.**

A last word on the badness of waits: An IBM "eBusiness poster" spotted outside the Benchmark Center in Rochester Minnesota contained this phrase:

All computers wait at the same speed.

Think about it.

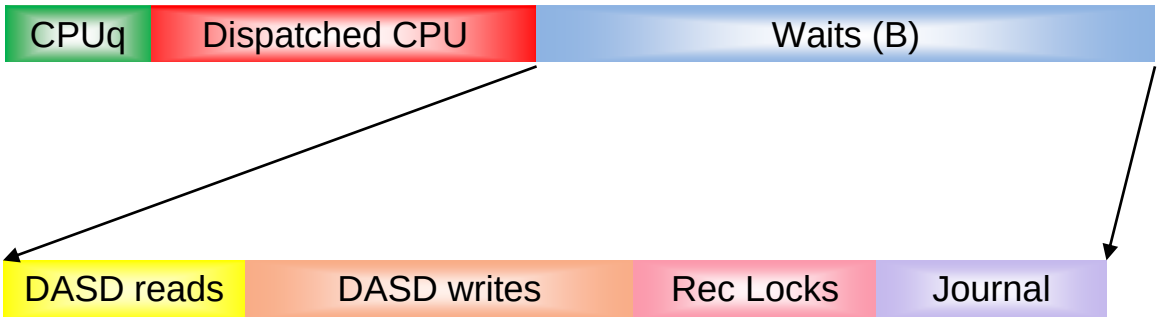
2.0 Detailing Waits

Up to here, this paper has made the case that wait analysis (and resulting "corrective actions") could lead to happiness. What is the first step in wait analysis? It begins with obtaining details on the individual waits.

Refresher: a summary Run/Wait Time Signature for a typical batch type job/thread might look like:

Wait analysis begins by bringing out details in the "Waits (B)" component.

For example:



This represents the first phase of detailing: the raw amount of time spent in different types of waits. The next obvious metric needed is the number of each type of wait:

DASD reads	DASD writes	Rec Locks	Journal
3,523	17,772	355	5,741

Computed averages are next. Suppose the durations / counts / averages were as follows:

DASD reads	DASD writes	Rec Locks	Journal
42s	73s	45s	44s
3,523	17,772	355	5,741
0.012s	0.004s	0.126s	0.007s

This is already enough information to begin contemplating actions. Some of the questions it raises include:

How many of the DASD reads are page faults? Would main memory/pool changes help?
What objects are being read from DASD?
What programs are causing the reads?
How could those DASD reads be reduced, eliminated, or made asynchronous?
Could the DASD read response time be better?
What objects are being written to DASD?
What programs are causing the writes?
How could those DASD writes be reduced, eliminated or made asynchronous?
Could the DASD write response time be better?
What DB2 files are involved with the record locks?
What programs are requesting the record locks?
What other jobs/threads are causing the record lock contention?
What files are being journaled?
What journals are involved?
Are the journals needed and optimally configured?
Could COMMIT cycles or the Journal PRPQ used to reduce this wait component?
Is the DASD I/O subsystem write cache(s) large enough?
Is the DASD configuration well balanced, in terms of IOPs, IOAs, busses, RAID configurations?

Unfortunately it is beyond the scope of this paper to delve into details of how to tackle the wait "corrective actions".

3.0 Wait Analysis on IBM i

All preceding material was a generic discussion of wait analysis. Now we'll focus on such capabilities that are built into IBM i with 6.1 and higher.

Remember back to the statement that a job/thread is either running on a processor, waiting for a processor to become available, or waiting for someone or something? The LIC has assigned an identifier to ALL[6] the points in LIC code that actually enter the wait state. [7] In V5R1 there were about 120 such wait points. In 6.1 there are over 260. Each individual wait point is sometimes referred to as an "enum". "Enum" is shorthand for the C++ programming language's "enumerated value" and simply means a fixed set of items. When a job/thread is in the wait state, it IS in one of the 260+ possible wait points. The "current wait" of a job/thread can be referred to by the numerical value of the "enum" (e.g. 51), or by a 3 character eye catcher that has been assigned to each enum (e.g. "JBw") or by a text string associated with each (e.g. "JOBUNDLEWAIT").

All jobs/tasks/threads on a system have instrumented wait 'buckets' which are used to keep track of the different types of waits each thread or task is experiencing over time. Because there are hundreds of different possible wait points known to the system we've reduced this down to a more manageable set of 32 wait buckets. Both Job Watcher and Collection Services Investigator surface these same wait bucket statistics (times and counts) for the 32 wait buckets found on the system.

3.1 Job Watcher

Job Watcher is a sampling based performance tool included with IBM i at 6.1 (see commands ADDJWDFN, STRJW.) At specified time intervals, or "as fast as possible", a STRJW will sample anywhere from 1 thread/job to all threads/jobs on a system. It gathers a large variety of performance data, much of it beyond the scope of this paper. But one of the main reasons for the creation of Job Watcher, was to capitalize on wait points first introduced into the system in V5R1.

3.2 Wait Buckets

A large number of individual wait points is great from a data-empowerment point of view. However, when it comes to keeping track of them on a wait-point by wait-point basis, for every unit of work, it presents challenges to efficient implementation. An ideal design would be for each of the possible wait points to have its own set of data associated with it, for each unit of work (job/thread/task). The minimum amount of accounting data that would be needed includes:

- Occurrence count
- Total time accumulator

---

[6] Some *types* of waits are identified with greater granularity than are other points. For example:. Locks and Seizes have more individual wait points identified than do other types of waits that tend to share block points.

[7] At the actual run/wait nitty gritty level, only LIC code can truly enter a wait. If an application or OS/400 program enters a wait state, it does so in LIC code it has caused to run on it's behalf.

It was determined that keeping 260+ pairs of these numbers associated with every job/thread/task on a machine was simply too much overhead (mainly in the area of main storage footprint).

A compromise was reached that allows for a potentially very large number of individual wait points to be mapped into a modest sized set of accounting data. The modest sized set of accounting numbers is called the Wait Buckets. There are 32 such buckets, but 3 of them have special purposes, so there are 29 buckets available to map the 260+ wait points. Again, these buckets exist on a per unit of work basis.

3.2.1 Do Wait Buckets defeat the purpose of many block points?

One might ask: "What's the value in having a large number of unique block points, if all this detail is going to be lost when they get crammed into 29 Wait Buckets?" That's a fair question. The real loss of granularity is felt with sampling based tools, like Job Watcher. But even with JW, there's good use of the high wait point counts:

At any given instant in time, the full granularity afforded by all the wait points is available to sampling based tools. For example: "At this particular moment in time, thread XYZ is waiting in block point enum 114. And it has been waiting there for n microseconds."

Trace based tools, e.g. PEX Analyzer, (which are beyond the scope of this paper) can "see" every wait transition, and effectively do the accounting on a per-enum basis, making full use of the granularity provided.

For these two reasons, maximizing wait point granularity is a good thing to do.

3.2.3 Wait Points ("enums") and Wait Buckets

As mentioned earlier, wait accounting is the core functionality of the Job Watcher tool. The LIC supports **remapping** of enums to buckets though this is very rarely done anymore at 6.1 and higher. At previous releases remapping the buckets was done by Job Watcher because the system was shipped with only 16 wait buckets used (instead of the 32 available). However at each new release, the bucket mapping could change.

Note: There were no changes to the buckets between 6.1 and 7.1 but there were changes to the buckets between 7.1 and 7.2.

The Wait Buckets defined on the system at 6.1 and 7.1 are:

1. Dispatched CPU
2. CPU queueing
3. Reserved
4. Other waits
5. Disk page faults

6. Disk non fault reads
7. Disk space usage contention
8. Disk op-start contention
9. Disk writes
10. Disk other
11. Journaling
12. Semaphore contention
13. Mutex contention
14. Machine level gate serialization
15. Seize contention
16. Database record lock contention
17. Object lock contention
18. Ineligible waits
19. Main storage pool overcommitment
20. Classic JVM user including locks
21. Classic JVM
22. Classic JVM other
23. Reserved
24. Socket transmits
25. Socket receives
26. Socket other
27. IFS
28. PASE
29. Data queue receives
30. Idle/waiting for work
31. Synchronization token contention
32. Abnormal contention

The Wait Buckets defined on the system at 7.2 are:

1. Dispatched CPU
2. CPU queueing
3. Reserved
4. Other waits
5. Disk page faults
6. Disk non fault reads
7. Disk space usage contention
8. Disk op-start contention
9. Disk writes
10. Disk other
11. Journaling
12. Semaphore contention
13. Mutex contention
14. Machine level gate serialization
15. Seize contention
16. Database record lock contention

17. Object lock contention
18. Ineligible waits
19. Main storage pool overcommitment
20. Journal save while active
21. Reserved
22. Reserved
23. Reserved
24. Socket transmits
25. Socket receives
26. Socket other
27. IFS
28. PASE
29. Data queue receives
30. Idle/waiting for work
31. Synchronization token contention
32. Abnormal contention

Additional details on the buckets and the enums that are assigned each follow.

3.3 The Wait Bucket Mapping for IBM i

Each of the 260+ block points in the system is some flavor of one of approximately 20 different LIC Queuing Primitives. Individual block points may be reported (i.e. assigned an enum) that is one of the Primitives' enums (which is the default assignment), OR (preferably) the specific block-owning LIC component can chose to "invent" another, more descriptive enum for the block point.

For example, synchronous DASD I/O READ wait. The author is not certain, but it is likely that the wait (block) that occurs in a job/thread while a synchronous DASD read is in progress is probably implemented with a LIC Queuing Primitive known as a "Single Task Blocker" (eye catcher QTB, enum number 4). That is, when LIC blocks a job/thread due to waiting for a synchronous DASD read to complete, it uses a QTB wait primitive/mechanism. If the component that owns this function (Storage Management) had done no further "IDing", that is how such waits would report (QTB, enum 4). That is OK, except there are probably a lot of other block points that ALSO use QTB. Therefore, it would be difficult/impossible to differentiate DASD READ blocks from other blocks. Fortunately, Storage Management, realizing how important it is to quantify DASD op waits, have invented a different eye catcher and enum (SRd, 158) that overrides QTB,4. Before you start to read this section on the Wait Buckets and their enums, you might want to read the description of Bucket 4 (Other Waits) first. Bucket 4 contains many of the default, LIC Queuing Primitives enums.

Disclaimer

The following discussion will include opinion. It will also, more than likely, be far less complete than many people (including the author) would like it to be. There's probably no single person that knows all the nuances of the 260+ wait points in IBM i. Also, in spite of 260+ individual points, many of these remain "general" and "generic" to some degree…preventing them from categorically being declared "normal/OK" or "bad". This discussion should be viewed as:

- Potentially in error
- Potentially out-of-date
- One person's opinion
- As a starting point, guideline to interpreting wait points and buckets, not as the "last/only word"

3.3.1 Bucket 1 – Dispatched CPU

This accumulates the amount of time a thread or task has been "dispatched" to a virtual processor. "Dispatched to a virtual processor" means the thread or task has been assigned a virtual processor so it can begin execution of machine instructions.

Note: Dispatched CPU is **not equal** to CPU utilization or CPU used time as normally seen in WRKJOB, WRKACTJOB, etc. Dispatched CPU time includes time dispatched to a virtual processor but not necessarily burning CPU cycles; it includes time sharing the virtual processor with other threads due to multithreading (SMT/HMT), time due to LPAR shared virtual processors, as well as time for memory and cache fetches, etc. Therefore Dispatched CPU is often much greater than the CPU used time because this CPU waiting/sharing time is included in Dispatched CPU. iDoctor has a breakdown graph that divides Dispatched CPU into 2 parts (active/waiting). Think of Dispatched CPU waiting as Dispatched CPU sharing if it makes you or your customer less concerned. This is normal system behavior!

Figure A depicts the benefits of two dispatched threads sharing a physical processor. Each thread's actual CPU processing is overlapped with the other thus keeping the processor busy. One thread can be executing instructions during the time where the other might be waiting upon a memory or cache fetch.

Dispatched CPU helps account for all of a job's elapsed time. The sum of Dispatched CPU plus CPU queuing plus the sum of the Wait time found in the other wait buckets will explain where time is being spent for the duration of the job/thread or for a time period under study or investigation. Substituting CPU for Dispatched CPU will generally cause the elapsed time result to be understated.

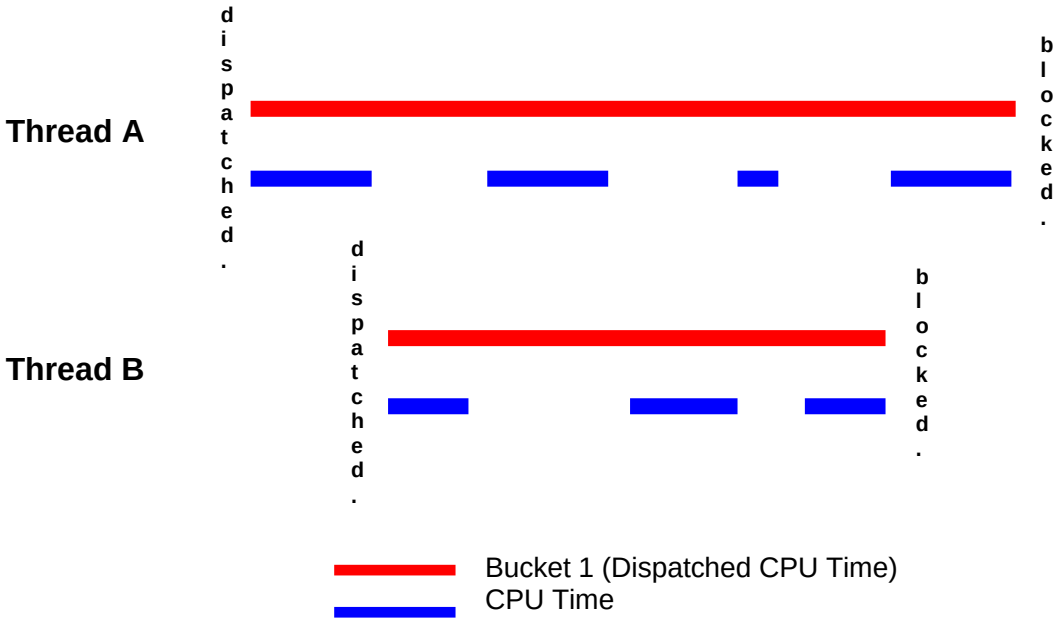


Figure A: The relationship between Dispatched CPU time and CPU time where two threads are sharing the same physical processor

The dispatched time value can differ from the "CPU Time" measured by other means (DSPJOB, WRKACTJOB, WRKSYSACT, job accounting, the DELTACPU field in Job Watcher itself). The difference can be large. The main factors that cause these discrepancies are:

1. Processor Hardware Multi Threading (HMT) feature. This can cause bucket 1's time to be larger than the actual CPU time. HMT is when more than one thread or task can be simultaneously assigned to the same physical processor. In that scenario, they share the processor's cycles, mainly during long "off chip" operations, like memory fetches. The Dispatched CPU bucket will record the elapsed time a thread or task has been dispatched. The real CPU value will only include the exact number of cycles used by the thread or task while it was dispatched.

2. Simultaneous Multithreading (SMT). This is used on POWER5+ processors and also causes bucket 1's time to be larger than the actual CPU time. Like HMT, multiple threads of execution can execute on the same processor at the same time although the mechanics at the processor and cache levels are different. On POWER7 processors there can be as many as four threads executing on the same physical processor at the same time. For a more thorough discussion of SMT and HMT, see the following white papers authored by Mark Funk: (1) Simultaneous Multi-Threading on eServer iSeries POWER5 (2) Simultaneous Multi-Threading on POWER7 Processors Further discussion and links to additional articles can also be found at Dawn May's i Can blog or the IBM i Performance Management website.

3. Background assisting tasks, like those used in the DB Multi-Tasking Feature. Background assisting tasks, which promote (add) their CPU usage back into the client job/thread, will cause the client thread's Dispatched CPU value to be smaller than the measured CPU time.[8]

4. LPAR shared/partial processors. This is where the tricky concept of Virtual Processors comes into play. The Dispatched CPU bucket actually records the elapsed time a thread or task is dispatched to a *Virtual Processor*, not (necessarily) a Physical Processor. Similar to HMT mentioned above, a Virtual Processor can be shared across LPAR partitions. If that occurs while a thread or task is dispatched to one of these, the bucket 1 time will be greater than the CPU time, because it will include time the thread/task is dispatched, but is "waiting for its turn" at the physical processor behind the virtual one.

3.3.2 Bucket 2 – CPU Queuing

No wait points assigned, waiting for a virtual processor is a special kind of wait. This is simply the number of microseconds a thread/task has waited… ready to run… for a virtual processor to become available. As stated above, a unit of work may have to wait for a physical processor even when in bucket 1 state.

Note: there may always be miniscule amounts of time reported in this bucket. It's an artifact of the fact that SOME tiny, finite amount of time transpires between when a thread becomes "ready to run" and when it is dispatched to a processor.

3.3.3 Bucket 3 – RESERVED

This bucket is not currently used. It may have numbers in it but its best to ignore them unless directed by IBM service.

3.3.4 Bucket 4 – Other Waits

The dreaded "other" word! Yes, even wait accounting must have a "catch all bucket". The enums assigned are:

Enum	Eye Catcher	Description
1	QCo	Qu counter
4	QTB	Qu single task blocker
5	QUW	Qu unblock when done, not otherwise identified
6	QQu	Qu queue, not otherwise identified
7	QTQ	Qu tree queue, not otherwise identified
9	QPo	Qu pool, not otherwise identified
10	QMP	Qu message pool, not otherwise identified
11	QMP	Qu simple message pool, not otherwise identified

[8] Conversely, the Dispatched CPU time value for the assisting tasks themselves would be larger (over time) than the CPU time as measured/seen by WRKSYSACT or Collection Services.

12	QSP	Qu stackless message pool, not otherwise identified
13	QSC	Qu state counter, not otherwise identified
17	QSB	Qu system blocker, not otherwise identified
18	QMC	Qu maso condition, not otherwise identified
19	QRQ	Qu resident queue, not otherwise identified
43	QCo	QuCounterServerReceive
143	Rsv	Seize/lock: service
201	JSL	JAVA: J9 wait for request response
240	RCA	LIC CHAIN FUNCTIONS: SMART CHAIN ACCESS
241	RCI	LIC CHAIN FUNCTIONS: SMART CHAIN ITERATOR
242	RCM	LIC CHAIN FUNCTIONS: CHAIN MUTATOR
243	RCB	LIC CHAIN FUNCTIONS: SMART CHAIN PRIORITY BUMP 1
244	RCB	LIC CHAIN FUNCTIONS: SMART CHAIN PRIORITY BUMP 2
245	RCE	LIC CHAIN FUNCTIONS: CHAIN ACCESS EXTENDED
246	RCX	LIC CHAIN FUNCTIONS: ADAPTABLE SMART CHAIN ACCESS
330	EMw	MI EVENT WAIT
342	QMo	OTHER MI QUEUE WAIT
406	DBX	DB default wait timer sleep

The above enums with eye catchers beginning with a ‘Q’ are the generic wait points… the low level LIC blocks that have not (yet) been uniquely identified. These enums will be seen when LIC code blocks that has not gone out of its way to uniquely identify the block point. The only identification that exists is the differentiation afforded by the type of LIC blocking primitive used. A few words/opinions can be offered for some of them:

QCo is frequently used for timed waits. The wait used at the core of the DLY JOB command is a QCo wait. It is also used by POSIX Condition Variable waits.

QTB is a wait primitive used for many purposes (unfortunately). About the only generic statement that can be made on it is that is used when a thread/task is waiting for a specific action to happen on its behalf… explicitly for THAT thread/task. For example, waiting for synchronous DASD reads and writes to complete use QTB blocks. Fortunately, DASD reads and writes have further been identified, so they are covered by their own unique buckets, they are not lumped into QTB (see later Buckets).

3.3.5 Bucket 5 – Disk page faults

These are the waits associated with implicit (page faults) DASD reads.

Page faults are frequently (but not exclusively) caused by having “too many jobs/threads running concurrently in too small of a main store pool”. If the faulted-on object type is a ‘1AEF’ (Temporary Process Control Space), then that is a likely cause. There are other types of activity though, where page faults are expected or “normal”:

- When a program or application first starts up in a job/thread.

- DB2 Access Paths (keyed parts of physical files, or Logical Files)… these tend to be referenced in a highly unpredictable way, and "faulting in" pages of access paths is considered "normal".
- Access pending faults (enum 165) refers to waits that did not, itself, issue a disk read, but is waiting on SOME in-progress disk read that might have been started asynchronously in this task/thread, or synchronously or asynchronsously in some other task/thread.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
161	SFt	MAINSTORE/LOGICAL-DASD-IO: PAGE FAULT
162	SFP	MAINSTORE/LOGICAL-DASD-IO: PAGE FAULT IO PENDING
164	GRf	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP READ FOR FAULT
165	SRR	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP READ FOR FAULT IO PENDING

3.3.6 Bucket 6 – Disk non fault reads

These are simply the waits associated with explicit ("read this from DASD for me") **synchronous** DASD reads.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
158	SRd	MAINSTORE/LOGICAL-DASD-IO: DASD READ
159	SRQ	MAINSTORE/LOGICAL-DASD-IO: DASD READ IO PENDING

3.3.7 Bucket 7 - Disk space usage contention

When an object, or internal LIC object is created or extended, and free DASD space has to be located to satisfy the request, there is some level of serialization performed. This is done on an ASP-by ASP and unit-by-unit basis. Normally, one would expect to see little, if any, of these types of waits. If they are present in significant percentages, it usually means the OS/LIC is being asked (by applications) to perform a very high RATE of object creates/extends/truncates or deletes. (Note: opening a DB2 file causes a create). The SIZE of the DASD space requests is not relevant to these blocks; it's the RATE of requests that is relevant.

The enums associated with this bucket are:

Enum	Eye Catcher	Description

145	ASM	DASD space manager: CONCURRENCY CONTENTION
146	ASM	DASD space manager: ASP FREE SPACE DIRECTORY
147	ASM	DASD space manager: RR FREE SPACE LOCK
148	ASM	DASD space manager: GP FREE SPACE LOCK
149	ASM	DASD space manager: PERMANENT DIRECTORY LOCK
180	ASM	DASD SPACE MANAGER: TEMPORARY DIRECTORY LOCK
181	ASM	DASD SPACE MANAGER: PERSISTENT STORAGE LOCK
182	ASM	DASD SPACE MANAGER: STATIC DIRECTORY LOCK
183	ASM	VIRTUAL ADDRESS MANAGER: BIG SEGMENT ID LOCK
184	ASM	VIRTUAL ADDRESS MANAGER: LITTLE SEGMENT ID LOCK
185	ASM	DASD SPACE MANAGER: IASP LOCK
186	ASM	DASD SPACE MANAGER: MOVE CHAIN
187	ASM	DASD SPACE MANAGER: HYPERSPACE LOCK
188	ASM	DASD SPACE MANAGER: NON PERSISTENT DATA LOCK
189	ASM	VIRTUAL ADDRESS MANAGER: TEMPORARY SEGMENT ID RANGE MAPPER LOCK
190	ASM	VIRTUAL ADDRESS MANAGER: PERMANENT SEGMENT ID RANGE MAPPER LOCK
191	ASM	VIRTUAL ADDRESS MANAGER: IASP SEGMENT ID RANGE MAPPER LOCK

3.3.8 Bucket 8 – Disk op-start contention

These waits occur when a DASD operation start is delayed due to a very high rate of concurrent DASD operations in progress at the moment it is requested.

Enum	Eye Catcher	Description
49	QRR	Qu res stack message pool, Abnormal DASD op-start contention

3.3.9 Bucket 9 - Disk writes

These are the waits associated with **synchronous** DASD writes, **or waiting for asynchronous DASD writes to complete**.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
167	SWt	MAINSTORE/LOGICAL-DASD-IO: DASD WRITE
168	SWP	MAINSTORE/LOGICAL-DASD-IO: DASD WRITE IO PENDING
170	SWp	MAINSTORE/LOGICAL-DASD-IO: PAGE OUT WRITE
171	GPg	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP PURGE
172	GPP	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP PURGE IO PENDING
174	GTA	MAINSTORE/LOGICAL-DASD-IO: GENERIC ASYNC IO TRACKER WAIT
175	GTS	MAINSTORE/LOGICAL-DASD-IO: GENERIC SINGLE TASK BLOCKER WAIT
176	GTT	MAINSTORE/LOGICAL-DASD-IO: GENERIC TIMED TASK BLOCKER

3.3.10 Bucket 10 – Disk other

This bucket includes waits for a variety of disk operations including the following:

- Waits that "mark disk locations" during Create, Extend, Truncate or Destroy of PERMANENT objects.
- Most bulk reads and writes performed during Save/Restore.
- Rarely seen waits found during disk unit configuration and setup.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
60	DSM	DASD management: find compression group
61	DSM	DASD management: deallocate compression group
62	DSM	DASD management: read compression directory
63	DSM	DASD management: write compression directory
64	DSM	DASD management: initialize compression start reorg
65	DSM	DASD management: mirror read sync
66	DSM	DASD management: mirror reassign sync
67	DSM	DASD management: mirror write verify sync
68	DSM	DASD management: read
69	DSM	DASD management: read diag
70	DSM	DASD management: verify
71	DSM	DASD management: verify diag
72	DSM	DASD management: write
73	DSM	DASD management: write diag
74	DSM	DASD management: write verify
75	DSM	DASD management: write verify diag
76	DSM	DASD management: reassign
77	DSM	DASD management: reassign diag
78	DSM	DASD management: allocate
79	DSM	DASD management: allocate diag
80	DSM	DASD management: deallocate
81	DSM	DASD management: deallocate diag
82	DSM	DASD management: enable auto allocate
83	DSM	DASD management: disable auto allocate
84	DSM	DASD management: query compression metrics
85	DSM	DASD management: query compression metrics diag
86	DSM	DASD management: compression scan read
87	DSM	DASD management: compression scan read diag
88	DSM	DASD management: compression discard temp data
89	DSM	DASD management: compression discard temp data diag
150	STv	MAINSTORE/LOGICAL-DASD-IO: SAR NOT SET
151	SRv	MAINSTORE/LOGICAL-DASD-IO: REMOVE

152	SRP	MAINSTORE/LOGICAL-DASD-IO: REMOVE IO PENDING
153	SCl	MAINSTORE/LOGICAL-DASD-IO: CLEAR
154	SCP	MAINSTORE/LOGICAL-DASD-IO: CLEAR IO PENDING
156	SUp	MAINSTORE/LOGICAL-DASD-IO: UNPIN
157	SUP	MAINSTORE/LOGICAL-DASD-IO: UNPIN IO PENDING
177	SMP	MAINSTORE/LOGICAL-DASD-IO: POOL CONFIGURATION
178	SMC	MAINSTORE/LOGICAL-DASD-IO: POOL CONFIGURATION CHANGE

3.3.11 Bucket 11 - Journaling

The waits associated with DB2 Journaling fall in this bucket.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
50	JBo	Journal bundle owner wait for DASD completion
51	JBw	Journal bundle wait for DASD completion
270	EFJ	EPFS: Wait for OS to finish apply journaled changes

Enum 50 is the wait in the thread that is actually performing the DASD write(s) to the journal. It is the wait for DASD Journal writes to complete. Journal uses some fancy approaches to DASD ops, to do their writes as efficiently as possible. That is why DASD writes to Journals do not fall in the "DASD Write" bucket below (this is a good thing for performance analysis, to have these Journal writes differentiated).

Enum 51 is the wait that occurs in threads other than the one that's performing the DASD write(s). For efficiency, multiple jobs/threads can "ride along" the journal DASD writes performed by other jobs/threads.

3.3.12 Bucket 12 - Semaphore contention

These are the block points used by C/C++ programming language (both operating system code, LPP and application code), usually in the POSIX environment, to implement Semaphore waits.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
16	QSm	Qu semaphore
353	Msw	Semaphore wait

3.3.13 Bucket 13 - Mutex contention

These are the block points used by C/C++ programming language (both operating system code, LPP and application code), usually in the POSIX environment, to implement Mutex waits.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
15	QMG	Qu mutex gate
350	Mmw	Mutex wait

3.3.14 Bucket 14 – Machine level gate serialization

The enums associated with this bucket are:

Enum	Eye Catcher	Description
2	QGa	Qu gate - high performance
3	QTG	Qu retry gate

QGa is a very high performance, low-overhead serialization primitive used by LIC. It is the type of primitive in which there can be one and only one "holder". Normally, QGa is used in areas in which the anticipated wait time, if any, is very small (microseconds).

Note: there are some related block points (QGb, QGc, QGd) that are later covered in the bucket named "ABNORMAL CONTENTION".

3.3.15 Bucket 15 - Seize contention

Think of seizes as the Licensed Internal Code's (LIC's) equivalent of Locks. A seize almost always occurs on/against an MI object (DB2 physical file member, Data Queue, Program, Library…). Seizes can conflict with Locks and can cause Lock conflicts. There is a large variety of seizes: shared, exclusive, "fair", and "intent-exclusive". It's beyond the scope of this paper to explain all there is to know about seizes. They are, after all, internal LIC primitives that are subject to change at any time. If seizes are a significant percentage of a Run/Wait Signature, examining the call stack, "wait object" and "holding task/thread" (if any) are probably necessary to understand what is causing the contention.

Seizes are frequently (but not exclusively) associated with data base objects and operations. Concurrent activities in multiple jobs such as opens, closes, journal sync points, access path building, etc might lead to seize waits. Other actions/objects that can experience seize waits include libraries and user profiles, during high rates of concurrent Create/Delete activity in multiple jobs.

This bucket was the first time that the term “holding task/thread” was mentioned. However, Job Watcher has that ability to determine the “holder” for more than just seize waits. It can do so for Locks, Data Base Record Locks and other wait enums based on a low level serialization primitive called a “gate”.

In the area of waiters and holders, it needs to be pointed out that the waiter… the job/thread that is experiencing the wait is frequently the victim, not the culprit.

The enums associated with this bucket are:

Enum	Eye Catcher	Description
100	Rex	Seize: exclusive
101	Rex	Seize: long running exclusive
102	Rsh	Seize: shared
103	Rix	Seize: intent exclusive
104	Ris	Seize: intent shared
105	Rfa	Seize: flush all
106	Rdx	Seize: database exclusive
107	Rii	Seize: internal intent exclusive
108	Rot	Seize: other
109	Rlk	Seize: lock conflict
112	RXX	Seize/lock impossible
125	Rsp	Seize: off-line IASP
126	Rra	Seize: release all
127	Rrs	Seize: release
133	Rss	Seize/lock: internal service tools hash class gate
135	Rmf	Seize: monitored free
141	Rcu	Seize: cleanup
320	SOo	COMMON MI OBJECT CHECKER: SEIZE OBJECT
321	SOi	COMMON MI OBJECT CHECKER: SEIZE FOR IPL NUMBER CHECK
421	Rsl	Seize: shared inhibit locks
422	Rfl	Seize: fair lock blocker

3.3.16 Bucket 16 - Database record lock contention

The enums associated with this bucket are:

Enum	Eye Catcher	Description
110	RDr	DB record lock: read
111	RDu	DB record lock: update
123	RDw	DB record lock: weak
134	Rxf	DB record lock: transfer
136	Rck	DB record lock: check
139	Rcx	DB record lock: conflict exit

A database weak record lock is only acquired thread-scoped and it only conflicts with update record locks which are thread-scoped to a different thread. The weak record lock does not conflict in any other situation. Weak record locks are used by SQE.

3.3.17 Bucket 17 – Object lock contention

These are the conflicts between threads involving objects. The OS frequently needs/obtains locks during such operations as:

- Opening a DB2 file
- Creating/deleting an object into a library
- Moving an object to a different library
- Ownership changes
- Etc

IBM i can also use "symbolic locks" as a serialization mechanism. These are called "space location locks".

Lastly, application code can explicitly use locks via the ALCOBJ CL command.

The enums in this bucket are:

Enum	Eye Catcher	Description
113	RIr	Lock: shared read
114	RIo	Lock: shared read only
115	RIu	Lock: shared update
116	RIa	Lock: exclusive allow read
117	RIe	Lock: exclusive no read
118	RMr	Lock: seize conflict
119	RMo	Lock: seize conflict
120	RMu	Lock: seize conflict
121	RMa	Lock: seize conflict
122	RMe	Lock: seize conflict
124	RMm	Lock: materialize
128	Rdo	Lock: destroy object
129	Rdp	Lock: destroy process
130	Rdt	Lock: destroy thread
131	Rdx	Lock: destroy TRXM
132	Rar	Lock: async retry
137	Rtr	Lock: trace
138	Rul	Lock: unlock
140	Rlc	Lock: lock count
142	Rpi	Lock: process interrupt

Note: the enums with the word "SEIZE" in the description are lock conflicts caused by existing seizes on an object.

3.3.18 Bucket 18 - Ineligible waits

This bucket simply quantifies the amount of time a thread has been in ineligible wait. A complete discussion of ineligible waits (and the control for it, "Max Active") is beyond the scope of this paper. But in general, if a system memory pool is configured with the correct maximum activity level, ineligible waits should not be occurring

The enums in this bucket are:

Enum	Eye Catcher	Description
280	WTI	RMPR: Wait to ineligible
281	ATI	RMPR: Active to ineligible

3.3.19 Bucket 19 – Main storage pool overcommitment

These waits indicate one or more main storage pools are currently overcommitted. Regular operations, like explicit DASD reads or page faults, are being delayed in order to locate "free" main storage page frames to hold the new incoming data.

The enums in this bucket are:

Enum	Eye Catcher	Description
155	GCP	MAINSTORE/LOGICAL-DASD-IO: CLEAR PAGE OUT WAIT
160	GRQ	MAINSTORE/LOGICAL-DASD-IO: DASD READ PAGE OUT WAIT
163	GFP	MAINSTORE/LOGICAL-DASD-IO: PAGE FAULT PAGE OUT WAIT
166	GRR	MAINSTORE/LOGICAL-DASD-IO: ACCESS GROUP READ FOR FAULT PAGE OUT WAIT
169	GWP	MAINSTORE/LOGICAL-DASD-IO: DASD WRITE PAGE OUT WAIT
173	SPw	MAINSTORE/LOGICAL-DASD-IO: PAGE OUT WAIT

3.3.20 Bucket 20

3.3.20.1 Bucket 20 – Classic JVM user including locks (6.1)

In 6.1, this bucket contains waits for jobs using IBM's Classic JVM. IBM's Classic JVM is not available after 6.1.[9]

For 6.1 the enums in this bucket are:

[9] The currently supported Java JVM is called IBM Technology for Java (J9). The waits caused by that JVM go into the PASE bucket.

Enum	Eye Catcher	Description
200	JUW	JAVA: USER WAIT
201	JSL	JAVA: USER SLEEP
203	JSU	JAVA: SUSPEND WAIT
209	JOL	JAVA: OBJECT LOCK
304	JSG	JAVA: SYNCHRONOUS GARBAGE COLLECTOR WAIT
305	JSF	JAVA: SYNCHRONOUS FINALIZATION WAIT

3.3.20.2 Bucket 20 – RESERVED (7.1)

3.3.20.3 Bucket 20 – Journal save while active (7.2)

For 7.2, the enums in this bucket are:

Enum	Eye Catcher	Description
52	JSW	Journal save while active wait

3.3.21 Bucket 21

3.3.21.1 Bucket 21 – Classic JVM (6.1)

This bucket contains waits for jobs using IBM's Classic JVM. IBM's Classic JVM is no longer available after 6.1.[10]

For 6.1 the enums in this bucket are:

Enum	Eye Catcher	Description
302	JWH	JAVA: GARBAGE COLLECTOR WAIT HANDSHAKE WAIT
303	JPH	JAVA: PRIMARY GC THREAD WAIT FOR HELPER THREADS DURING SWEEP
306	JGW	JAVA: GARBAGE COLLECTOR WAITING FOR WORK
307	JFW	JAVA: FINALIZATION WAITING FOR WORK
308	JVW	JAVA: VERBOSE WAITING FOR WORK

3.3.21.2 Bucket 21 – RESERVED (7.1/7.2)

[10] The currently supported Java JVM is called IBM Technology for Java (J9). The waits caused by that JVM go into the PASE bucket.

3.3.22 Bucket 22

3.3.22.1 Bucket 22 - Classic JVM other (6.1)

This bucket contains waits for jobs using IBM's Classic JVM. IBM's Classic JVM is no longer available after 6.1.[11]

For 6.1 the enums in this bucket are:

Enum	Eye Catcher	Description
202	JWC	JAVA: WAIT FOR COUNT
204	JEA	JAVA: END ALL THREADS
205	JDE	JAVA: DESTROY WAIT
206	JSD	JAVA: SHUTDOWN
207	JCL	JAVA: CLASS LOAD WAIT
208	JSL	JAVA: SIMPLE LOCK
300	JGG	JAVA: GARBAGE COLLECTOR GATE GUARD WAIT
301	JAB	JAVA: GARBAGE COLLECTOR ABORT WAIT
309	JGD	JAVA: GARBAGE COLLECTION DISABLE WAIT
310	JGE	JAVA: GARBAGE COLLECTION ENABLE WAIT

3.3.22.2 Bucket 22 – RESERVED (7.1/7.2)

3.3.23 Bucket 23 – RESERVED

3.3.24 Bucket 24 - Socket transmits

These are waits associated with Socket API calls that are sending/transmitting data.

The enums in this bucket are:

Enum	Eye Catcher	Description
212	STS	COMM/SOCKETS: SHORT WAIT FOR TCP SEND
213	LTS	COMM/SOCKETS: LONG WAIT FOR TCP SEND
216	SUS	COMM/SOCKETS: SHORT WAIT FOR UDP SEND
217	LUS	COMM/SOCKETS: LONG WAIT FOR UDP SEND

3.3.25 Bucket 25 - Socket receives

These are waits associated with Socket API calls that are receiving data.

The enums in this bucket are:

[11] See previous

Enum	Eye Catcher	Description
214	STR	COMM/SOCKETS: SHORT WAIT FOR TCP RECEIVE
215	LTR	COMM/SOCKETS: LONG WAIT FOR TCP RECEIVE
218	SUR	COMM/SOCKETS: SHORT WAIT FOR UDP RECEIVE
219	LUR	COMM/SOCKETS: LONG WAIT FOR UDP RECEIVE

3.3.26 Bucket 26 - Socket other

The primary wait points that should be seen from this bucket involve the SELECT socket API. That API can be used by an application for a variety of complex waiting scenarios.

The enums in this bucket are:

Enum	Eye Catcher	Description
220	SAS	COMM/SOCKETS: SHORT WAIT FOR IO COMPLETION
221	LAS	COMM/SOCKETS: LONG WAIT FOR IO COMPLETION
222	SSW	COMM/SOCKETS: SELECT SHORT WAIT
223	SLW	COMM/SOCKETS: SELECT LONG WAIT

3.3.27 Bucket 27 – IFS

These waits are due to Integrated File System (IFS) "pipe" operations.

The enums in this bucket are:

Enum	Eye Catcher	Description
250	PRL	IFS/PIPE: File table entry exclusive lock
251	PRC	IFS/PIPE: LIC reference count
252	PPC	IFS/PIPE: Main pipe count
253	PRP	IFS/PIPE: Read end of pipe
254	PWP	IFS/PIPE: Write end of pipe
255	PRW	IFS/PIPE: Pipe read waiters
256	PWW	IFS/PIPE: Pipe write waiters
257	PR1	IFS/PIPE: Read data lock 1
258	PR2	IFS/PIPE: Read data lock 2
259	PW1	IFS/PIPE: Write data lock 1
260	PW2	R610 - IFS/PIPE: Write data lock 2
261	PW3	R610 - IFS/PIPE: Write data lock 3
262	PW4	R610 - IFS/PIPE: Write data lock 4
263	PS1	IFS/PIPE: Stat lock
264	PA1	IFS/PIPE: Set attribute lock

265	PP1	IFS/PIPE: Poll lock
266	PA1	IFS/PIPE: Add reference lock
267	PL1	IFS/PIPE: Release reference lock

3.3.28 Bucket 28 - PASE

This bucket contains waits for PASE (Portable Application Solutions Environment). PASE is a solution that allows AIX applications to be ported to IBM i. Java applications using the new J9 JVM (IBM Technology for Java) will have their wait times shown as one of these PASE waits.

The enums in this bucket are:

Enum	Eye Catcher	Description
360	U60	PASE: fork
361	U61	PASE: msleep
362	U62	PASE: nsleep
363	U63	PASE: pause
364	U64	PASE: private tsleep event
365	U65	PASE: private wait lock
366	U66	PASE: ptrace PT attach
367	U67	PASE: ptrace PT delay att
368	U68	PASE: ptrace ttrcsig
369	U69	PASE: ptrace target
370	U70	PASE: sig suspend
371	U71	PASE: thread set sched
372	U72	PASE: thread set state
373	U73	PASE: thread set state fast
374	U74	PASE: thread tsleep
375	U75	PASE: thread tsleep event
376	U76	PASE: thread wait lock
377	U77	PASE: thread wait lock local
378	U78	PASE: core dump
379	U79	PASE: thread stopped
380	U80	PASE: run PASE thread
381	U81	PASE: run PASE thread attach
382	U82	PASE: termination serializer
383	U83	PASE: wait for exit
384	U84	PASE: PDC kernel map
385	U85	PASE: PDC prepare module
386	U86	PASE: close
387	U87	PASE: wait PID
388	U88	PASE: loader IPL
389	U89	PASE: loader lock

390	U90	PASE: ptrace lock

3.3.29 Bucket 29 - Data queue receives

These are the waits on MI Data Queue objects.

The enums in this bucket are:

Enum	Eye Catcher	Description
341	QMd	DATA QUEUE WAIT

3.3.30 Bucket 30 - Idle/waiting for work

These waits generally reflect an application that is either idle or waiting for additional work to perform for the user.

For legacy applications, such as 5250 I/O, this bucket truly does represent "idle time". However, for more modern applications, that might not be true. The time spent in this bucket might indicate a problem somewhere in the partition, or perhaps in some outboard system/LPAR COMM attached. For example, a thread might be waiting on a Socket Accept... waiting for a new unit of work to arrive. If some external issue was preventing the work from arriving, this is where the delay would be accounted for. Likewise, if a thread is waiting for work from another thread in the same partition, the same would be true, but in that case perhaps the wait buckets in the other thread would indicate the root of the problem.

The enums in this bucket are:

Enum	Eye Catcher	Description
37	Gai	QuGate idle
38	TGi	QuGate idle retry
39	MGi	QuMutexGate idle
200	JUW	JAVA: J9 wait for request
210	STA	COMM/SOCKETS: SHORT WAIT FOR ACCEPT
211	LTA	COMM/SOCKETS: LONG WAIT FOR ACCEPT
271	ERJ	EPFS: Wait for OS request to apply journaled changes
340	QMr	IDLE WAIT
351	Mcw	Condition wait

Enum 340 contains waits on a MI queue associated with each OS job known as the “MI Response Queue”. Normally, for 5250 type interactive applications, this would reflect the key/think time. Other possible uses would be APPC/APPN SNA type communications waits.

3.3.31 Bucket 31 - Synchronization token contention

These waits are a special type of wait used by C/C++ applications.

The enums in this bucket are:

Enum	Eye Catcher	Description
352	Mtw	Synchronization token wait

3.3.32 Bucket 32 – Abnormal contention

These waits reflect a high rate of concurrent waits/releases occurring against a wide variety of many of the other wait points listed previously. There are two types of these waits:

a. Unsuccessful wakeup retries (QGb, QGc, QGd)
b. Waiting in line to buy a ticket that gets you into the main wait line (QWL)

The enums in this bucket are:

Enum	Eye Catcher	Description
8	QRP	Qu res stack message pool
14	QWL	Qu wait list - waiting for access to a wait list (abnormal contention)
40	QGb	QuGateB
41	QGc	QuGateC
42	QGd	QuGateD
192	TLB	ASM TLB throttle segment destroy
193	TLB	ASM TLB throttle reserved
400	DMS	DB monitored seize timer sleep - Indicates long held seize in query that is monitored by some request to get an exclusive seize.
401	DEC	DB enforce constraint timer sleep
402	IMS	DB index build monitored seize timer sleep
403	CAS	Common function atomic update timer sleep
404	IMC	DB index build message cleanup timer sleep
405	IZE	DB index size estimate timer sleep

Trademarks and Disclaimers

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Photographs shown are of engineering prototypes. Changes may be incorporated in production models.