0 Change History

0.6.0 Initial Printed Release

Document as printed 12 Dec 2018

0.6.1 Initial PDF Release

Small change delta from SAFE[0.8.0] in BAST section

0.6.2 PDF Release

Considerable change to the GAST and BAST sections, roughly edited to page 43.

0.6.3 PDF Release

Most pages changed. Still too much change ongoing for change bars to be useful.

0.6.4 PDF Release

General restructuring of document, Hopefully it is starting to make some sense to the uninitiated.

0.6.5 PDF Release

0.8.4 probably not as good for the uninitiate as hoped. More major block movement, more detail added, Process Address Space and Tag (PAS and PAST) changed to Data Address Space (DAS) and Data Address Space (DAS). Hopefully better.

0.6.6 PDF Release

Edited from beginning to end, First consumable release.

0.6.7 PDF Release

Restriction of AAST to 50 bits and CAST to 20 bits.

0.6.8 PDF Release

Significant update of Storage Structure and other LLS data structures.

0.6.9 PDF Release

More changes to Storage Structure and other LLS data structures. This document version is not yet stable with ongoing sections being changed.

0.6.10 PDF Release

Still more changes to Storage Structure and embedded LLS system operational model. This document version is starting to become stable, though changes are still being made on edges. Major document refactoring suggested by Dr. Sohan not yet undertaken.

0.6.11 PDF Release

Largely repaired BAST/GAST interaction and structures (a BAST index no longer have any relation to any of its associated GAST indices).

0.7.0 PDF Release

Reintegrated file and data into single data object store. Edit of entire document.

0.7.1 PDF Release

Fracturing of doc into SAFE Architectural Reference Manual and SAFE Reference Implementation Manual, the latter barely a document.

0.7.2 PDF Release

New expansion model via adding 2 SR MSBs per PR MSB. expansion of underspecified SAE model.

Review Notes (still not integrated into document):

MN return to sender if MN target queue is full.

1 Introduction

Secure Architectural Framework Enhancements (SAFE) employs a small change in computational architecture to effect a significant change in the architecture's security model. Its primary goal is to significantly change the operational security mechanism for computing in general (sensor-controllers to supercomputers), while preserving application software, minimizing operational cost, and maximizing performance. SAFE offers compatibility with legacy application binaries, that it is faster, lower power, and simpler than the legacy architecture it replaces; and that most importantly is ***secureable***[1].

SAFE is a hardware-based containerization technology that replaces the lowest levels of fungible Operating System (OS) software with hardware. SAFE's primary security mechanism is the universal application of a hardware managed, hierarchically cached, *Address Space* (AS) model that relies on indices (tags) to select one *Address Space Tag* (AST) from a plethora of concurrently accessible ASTs. An AS is very similar to a 1970's era segment, with a number of notable exceptions[2]. SAFE abandons the legacy term *segment* in favor of the term *Address Space*. Segmentation's initial meaning has been clouded by a general incongruity between its initial Multics notion (a security and management focused concept) and Intel's usage relative to its 8086 architecture (an address range extension mechanism); and because of a fairly common and somewhat uninformed rejection of segmentation's value proposition that grew out of the wide-spread adoption of Unix's demand-paged memory model.

On one level, SAFE is a hardware implementation of some of the more security centric aspects of Multics[3], but only in the loosest sense. Multic's notion of merging memory and file access via segmentation dates back to the mid 1960s. The hardware in this era was very limited and the notion of a 64-bit address was still almost three decades away from reality. SAFE uses hardware based memory/storage partitioning to create ASs.

1. SAFE relies on software for the configuration of its security boundaries and as such its inherent security mechanisms cannot transcend its configuration. SAFE cannot replace a necessary, externally supplied (system owner), customization capability (configuration). System software ultimately controls which users can access which data objects. Once an access decision is made, SAFE can provide an iron-clad guarantee that the configured access rules cannot be circumvented, but poorly configured systems can be made insecure. Properly configured SAFE systems have the property of being internally "air gapped", that is, SAFE allows a level of compartmentalization/containerization within a system that has security characteristics that are indistinguishable from air-gapped independent systems. Security concerns no longer necessitate the incorporation of independent systems when the processing/storage capabilities of a single system are otherwise sufficient (this has the added benefit of decreasing the infrastructure costs associated with multiple systems).

2. 1970's ere segments were a software managed memory/storage construct. ASs are a mostly hardware managed data system that transcend traditional software (operating system) control.

3. SAFE and Multics are fundamentally different, but some key attributes of Multics are present in SAFE in ways that they are not present in legacy systems. Conversely, Multics' most enduring innovation, the hierarchical file system, is not present in SAFE. Multics is recognized for its place in history and this footnote was added because individuals familiar with computational architecture and operating system history tend to gravitate to SAFE's Multics parallels. Without acknowledging these similarities and giving Multics its due, it can become difficult for some to move past what we term the "Multics Issue" and to focus on the characteristics of SAFE. None of the segmentation (Multics) discussion is advantaged by the fact that Multics was quickly displaced by Unix (whose very name was a "dig" at [perceived] Multics' complexity), and hence, in some circles, Multics is held in a certain disrepute. This is probably, in part, because of the significant regard in which Thompson, Ritchie, and Kernighan are held, and the sense that if they did not like it, then who are we to disagree. It is also true, that in its time, Multics introduced a level of complexity that Unix avoided. In summary, SAFE salutes Multics conceptual contributions, and like every OS designed since 1970, SAFE incorporates aspects of Multics, but the similarities are largely conceptual.

A Data AS (DAS) is a data object that has an associated 50-bit access range. Within SAFE, thousands of DASs are concurrently and independently accessible by each process via their respective DASTs. This AST vector supports the selection of a particular data object from the set of available data objects. SAFE process addresses are 64-bit 3-tuples (Type, AST, Offset) in which: a *Type* indicates with which function-specific, system level AST vector the associated reference is associated, an *AST* indicates which data object within the type-specific vector reference is associated, and an *Offset* specifies where within the referenced data object the associated access is to take place. A 50-bit DAS Offset range limit is logically, but not functionally restrictive (many systems have data objects that are logically more than 1pB in size), but data objects of such extreme size are broken into shards[1] for management purposes and they employ object-type specific access interfaces that unify their physically diverse shards into a logically monolithic data object. SAFE's data object size limit makes SAFE's compatible with, and comparable to, legacy system's data management organizations. A simple extension of SAFE's existing mechanism supports the coalescing of aligned, contiguous subsets of 1pB ASs into larger (up to 1exbibyte) data objects, but doing so does not at present seem necessary.

On another level, SAFE is a hardware implementation of a microkernel, using ASs in place of *ports*. SAFE's hardware implementation of the core operational system management responsibilities (memory management, file management, device management, and process management) makes SAFE hardware somewhat synonymous with core microkernel software subsystems, from a functionality perspective. SAFE's AS mechanism is a direct hardware implementation of the microkernel port concept (frequently obviating the need to copy data across the port interface). SAFE provides both security and performance, issues that have historically been mutual exclusively associated with microkernels.

SAFE represents a hardware supported confluence of Multics-ish merged memory/file segmentation and hardware microkernel operational (with persistent storage management elevated to core OS status). SAFE adopts a modern view of the hardware associated with a memory hierarchy (employing caching as the mechanism for leveraging the capabilities of multiple levels of a merged memory-storage subsystem) and a costing model consistent with the realities of 21st century system component costs (CPUs are commodititized, easily accessible, plentiful, and increasingly inexpensive; the hardware infrastructures of memory and storage are merging; and electricity and physical infrastructure are becoming relatively more expensive than the systems with which they are associated). SAFE incorporates the dicta: "Security is not about what is supposed to happen, it's about what does happen, and "hardware is molded from sand and software is cast in concrete"[2]. Security is a software problem, but software cannot, to any significant degree, factor into the solution. Complexity is at the root of the security problem and attempting to address it by adding software increases complexity and exacerbates the problem. SAFE's underlying operational model is simpler, faster, and lower power than the legacy mechanism it replaces.

1. The 50-bit, per AS address range is deemed to be at the limit of reasonably prudent, individual, physical data object size, with no room for future expansion. However, a reasonably cogent argument can be constructed that the 1pB data object size limit is somewhat larger than necessary to support the sharding that physical storage management employs. It is also still effectively larger than the functional limits of current OSs. For example, although Microsoft Windows 10 theoretically supports 64-bit file sizes, in practice the limit is 48 bits, up from a 44-bit limit from a decade earlier).

2. This aphorism is, as far as we know, unique to the SAFE effort and a crucial aspect of its potential. Data is important. Software is useful. Hardware is functionally nothing more than a support system for software. Software is the backdrop against which data exists. Data is the value component of most computation. Data has persistent value. Software's value is associated with its ability to add value to data. Hardware's value is its ability to run software. Hardware is continually replaced. Hardware need only preserve its compatibility with an existing software model for its value proposition to shift to secondary considerations, of which security must now be considered paramount.

2 Identification and Authentication

SAFE employs **Object Identification Tuples (OITs)** to identify its computational objects (data, code, users, and hardware components) and **Object Authentication Tuples (OATs)** to guarantee the authenticity of OITs. Both OITs and OATs are 64 bytes in size. OITs can be used by sources external to a system to identify objects within a system. OITs are exclusively generated by their associated system, there is no mecha-nism via which an OIT can be initialized externally. When appropriate, a system may link a computational object to an OIT. OATs are always purely managed internally and they are never exposed to users, either directly or via programs.

An OIT contains 3 fields:

1) 448-bit Key (random, unique within a system assigned Domain)
2) 32-bit Domain (assigned external identificaiton)
3) 32-bit Value. (assigned internal reference enumerations)

An OIT exists with a vector of ~4 billion Domains. a Key is Domain specific and a Key is not guaranteed to be unique across Domains. A Value is the OIT's referent. It typically references a sys-tem's internal structure and is used as an enumerable reference (one that is more manageable than a 480-bit number). OIT Key field values are randomly assigned with guaranteed uniqueness within a system assigned Domain (uniqueness is not guaranteed between Domains, but implementations are free to assign Domain agnostic uniqueness to Keys). For a given random Key value within a specific Domain, a sys-tem will assign a Value (Keys are randomly generated, Domains and Values are assigned).

An OAT contains 2 fields: a 256-bit Key and a 256-bit Value. The Key is randomly assigned and is typically the encryption/decryption key for an associated object. The Value is typically the authentication checksum for the associated object's encryption/decryption and is thus generated as an artifact of SAFE's encryption/decryption process. An OAT is only useful in association with another object. It is typically the case that an **OIT's Value identifies a system internal object, which has an associated OAT**. Hence, SAFE objects are typically triplets of OIT, OAT, and a computational object (typically data or code). Large data objects cannot rely on a single authentication checksum (the entire data object would need to be decrypted to guarantee the integrity of a small portion of its contents). Large data objects typically rely on the object's OAT Key and smaller (64-bit or 128-bit), per block checksums (which are folded (Xor'ed) into the OAT's 256-bit value).

Table 1: Object Identification Tuple Format

Word	63	62	61	...	34	33	32	31	30	29	...	2	1	0
0	Key [63: 0]													
1	Key [127: 64]													
2	Key [191:128]													
3	Key [255:192]													
4	Key [319:256]													
5	Key [383:320]													
6	Key [447:384]													
7	Domain							Value						

Table 2: Object Authentication Tuple Format

Word	63	62	61	...	34	33	32	31	30	29	...	2	1	0
0	Key [63: 0]													
1	Key [127: 64]													
2	Key [191:128]													
3	Key [255:192]													
4	Value [63: 0]													
5	Value [127: 64]													
6	Value [191:128]													
7	Value [255:192]													

3 Processing Model

A SAFE system is composed of seven major component types:

- **Caches:** SAFE relies on a Cache Hierarchy (CH).
- **Last-Level Store (LLS):** An LLS anchors a CH. It provides the back-ing store for a system's DSASs and manages the lowest levels of a system's data object security protocol.
- **Process Management Unit (PMU):** A PMU manages SAFE's auto-mated process management functions and is associated with Level-Four (L4) of the CH. The PMU oversees a SAFE system's process management, one of the 4 OS functions subsumed by SAFE Virtual Hardware[1] (SVH). PMUs also manage interprocess communication and network connections.
- **Secure Networking Engine (SNE):** a monolithic processing capabil-ity associated with complete user level support (no intervening OS mediation is required) of a networking interface (for example: Ether-net, SONET, and SDH), but that can also be associated with other interfaces not traditionally considered to be networking (for exam-ple: USB, Thunderbolt, PS/2, RS232) that transit information in and/or out of a system. An SNE represents a generalized communication interface that supports its physical transport layer(s) at the system call level, supporting direct interaction between an SNE and an application running on a CPU core. A system will typically have two or more SNEs, with the ultimate SNE count determined by the num-ber of external interfaces and the number of interfaces abstracted by its SNEs.
- **Secure Accelerator Engine (SAE):** a shared acceleration resource that performs a set of standard processing tasks on behalf of a pro-cess via a standard **Remote Procedure Call (RPC) interface**. The remotely triggered functions can be arbitrarily complex. The princi-pal distinction between a **Processing Node (PN)** that executes user code and an SAE is the degree of control exerted by a user. SAEs are concurrently shared (many processes can interact with an SAE at the same time), relative to a serially shared PN. A GPU-like standard graphics interface in which the SNE has autonomous control over a display and a process requests display services from an SAE is a good example of an SAE. An encryption engine, through which a process can send data blocks to have them encrypted or decrypted via a process supplied key is another example of an SAE.
- **Authentication Engine (AE):** a processing subsystem that manages user authentication via a collection of internal data and external I/O devices. The proposed authentication model relies on visual data, auditory data, wireless/radio-frequency/infra-red/WiFi/ etc. devices that are carried by users, and user relative and/or known user-based information. An AE employs its user authentication mechanisms to map its external sensors input to a users OIT. A system can be designed with an AE that returns an OIT to its associated system, which matches system-based OATs with the AE-based OIT. Con-versely, an AE can use two independent identity recognition mecha-nisms that independently map to OITs and the OITs can be compared to each other before being authenticated by its OAT. The dual-map model can be extended to a multi-mapped mechanism and independent OITs can be pooled, with an A though C and X of Y authentication mechanism (OITs A, B, and C must be present and 2(X) of 4(Y) additional OITs must be present to authenticate a user)
- **Processing Node (PN)**: a SAFE system supports a range of process-ing capabilities. In all cases a PN is abstracted by a SAFE Adapta-tion Layer and a PMU manages them in a largely undifferentiated manner: a PN is associated with a C**ontext Address Space (CAS)** and its access to system resources is completely controlled by the state thereof. SAFE PNs are:
  - **Central Processing Unit (CPU):** for our purposes each individual processor (core) in a legacy CPU is a PN.
  - **User Controlled Accelerator (UCA):** a program/thread execution engine that is functionally a processing unit that executes user supplied code, but that is not a traditional CPU. An FPGA that is user programmed and controlled is an example of a UCA. A UCA may have a different context switch model than a CPU, but

1. Hardware is becoming softer, with embedded cores and software/firm-ware handling duties that previously have been performed by logic gates.

any differences are abstracted by the UCA's SAFE Adaptation Layer and UCAs are treated like CPUs with potentially modified scheduling mechanisms.

- **Hybrid Processing Units (HPU):** the combination of multiple CPUs and/or UCAs behind a single scheduling interface constitutes an HPU. From a PMU's perspective this is a single schedule-leable unit that executes using the standard SAFE Adaptation Layer. The SAFE interface protects the associated system from the operation of an HPU. A CPU with a captive GPU or vector of GPUs is an HPU. PMU power management is a heightened con-cern when dealing with HPUs (individual CPU cores ans acceler-ation cores require power management as well, but more extensive hardware resource blocks have greater power con-sumption capabilities.)

Processing-in-Memory (PIM) has become a topic of significant interest. It is generally associated with the movement of processing into a sys-tem's memory hierarchy. SAFE adopts a PIM model that migrates secu-rity sensitive software functionality out of standard processing cores and into embedded systems that are placed in the memory hierarchy. This does not make SAFE less applicable to standard PIM models, it makes SAFE a more secure model for other PIM applications.

4 System Interconnect / Cache Protocol

SAFE's system interconnect is a Cache Hierarchy (CH) with associated cache-level to cache-level connections and protocol. The protocol between the layers is comprised of two basic communication models, data and message, with a small cross-pollination of the message protocol into the data layer. In general, the cache protocol looks like a vanilla PN->LLS Request[1], LLS->PN response model, with downstream inval-idation/write-through and upstream response. There are simple optimizations that allow an upstream node to write-through partial cache lines, but this is a simple bandwidth optimization of a write-through request or response. It does not represent a departure from the basic protocol (except when an optionally supported, network-specific, buffer optimi-zation is enabled, which necessitates active cache line merging, which results from by a cache line being subjected to both a socket source and target being capable of writing to it).

1. Historically, a occasionally inconsistent upstream/downstream terminology has been applied to hierarchical communication. Within SAFE, *PN->LLS* terminology indicates data/packet movement from a higher-level cache (a cache level closer to a Processing Node (PN)/CPU) to a lower-level cache (a cache closer to the Last-Level Store (LLS)/disk). *LLS->PN* terminology indicates data/packet movement from a cache level closer to a LLS to a cache closer to a PN. Data direction is decorated with the contents/type of the data being moved. A request is an independently originated data element that will typically have a response. In general terms requests and responses flow in opposite directions, but SAFE does allow for direct peer-to-peer communication, which is labeled as a PN->PN flow (independent of the cache level at which the communication occurs), with *request* indicating the instigator of a transaction and *response* indicating resolution of a requested transaction. For example, a PN requesting a Level-One Cache (L1) line would issue a *PN->LLS Request*. and the response from the neighboring Level-Two Cache (L2) would be an *LLS->PN Response*. Conversely, an LLS that needs to flush a line from a cache would issue an *LLS->PN Request*, and its neighboring Level-Five Cache (L5) would issue a *PN->LLS Response*, when the request has been satisfied.

Cache protocol requests and their associated data centric responses migrate from cache-level to cache-level. Logically, a cache makes a request and a neighboring cache responds to the request. This can be highly transitive with requests flowing up or down a CH until they reach a cache at which affirmative action can be taken, with responses then transitively flowing back down or up the CH until they reach the level as which associated request originated.

If a lower-level cache cannot respond to a request from a neighboring higher-level cache in a timely manner, it may respond to the request with a Delayed-Response. This delay can be open-ended or timed (a cache can respond with an expected availability delta/delay or not). **Open-Ended Delayed-Responses** indicate that the requesting PN's data response time is indeterminate. **Timed Delayed-Responses** indicate that a data response cannot occur in a timely fashion, but that the data will probably be available within the indicated time-frame. A request indi-cates whether the data associated with a request should be speculatively push up the CH as it becomes available. Both Open-Ended and Timed delays place the onus on the requesting process to re-issue the request (the Delayed-Response terminates the request). No state is maintained for the previous request within the cache hierarchy, aside from the terminating device (typically an LLS or Secure Networking Engine (SNE)). Open-ended delayed responses require that a cache also notify the Process Management Unit (PMU) that an Open-Ended Delayed Response messages has been sent, as do Timed Delayed-Response mes-sages with timer valued that exceed the system's Wait-for-Response time delta.

Ignoring the details associated with a lower-level data integrity proto-col, the SAFE Cache Hierarchy Protocol (CHP) involves (not exclu-sively):

- **Line Requests from higher-level caches that are matched to Line Responses from lower-level caches.** This is triggered in the higher-level cache by a cache miss. The higher-level cache asks the lower-level cache for the missing data. The lower-level cache responds with the data when it becomes available. If the lower-level cache

does not expect to be able to supply the requested data in a timely fashion, a *Delayed* response is returned. An invalid request is indistinguishable from a heavily delayed request (the associated process is swapped out and when it is out of the PN, it is killed by its associated PMU).

- **Line Invalidation / Line Discard requests from lower-level caches are matched to Write-Through Response and/or Eviction Responses from higher-level caches.** The lower-level cache needs to evict (withdraw) a cache line from a higher-level cache. If the cache line is dirty it sends a write-through and evict request; and the higher-level cache sends a write-through with invalidation response. If the cache line is clean, the lower-level cache sends an invalidation request and the higher level cache sends an invalidation response.
- **Line Write-Through Requests from higher-level caches are matched to responses from lower-level caches**. A higher-level cache wishes to clean a dirty cache line and sends a write-through request. On receiving a write-through response from the lower-level cache, the higher-level cache transitions the associated cache line from dirty to clean.
- **Line discard requests from higher-level caches are matched to Line Discard Response from lower-level cachse.** A higher-level cache wishes to discard a cache line. The line is clean in the higher-level cache and can be discarded. A message is sent to the lower-level cache to facilitate management of the associated cache line's state.
- **A Line Requests from higher-level cache (Cache A) is tied to a transitive request from its lower-level cache (Cache B) to a peer cache of A (Cache C), then to a peer-to-peer response from C to A.** A higher-level cache requests a line from a lower-level cache. The associated line is dirty in a peer to the requesting cache, or it is clean in a peer to the requesting cache and not present in the lower-level cache (represented by a Null Cache line that has tag state, but no data). The lower-level cache cannot respond to the request, but knows a peer cache to the requesting higher-level cache that can. The lower-level cache sends a forwarding request to a peer of the requesting higher-level cache. The peer higher-level cache forwards the requested cache line to the original requester,
- **Line Repositioning Requests from lower-level caches matched to Line Repositioning Responses from higher-level caches.** A lower-level cache is going to move a cache line to a Victim Cache or a Null Cache. Before the line can be relocated in the lower-level cache, its new target in the lower-level cache must be communicated to all its associated higher-level caches. (This mechanism is associated with a cache access optimization that allows a higher-level cache to know the location of its lines in the next lower-level cache, it is not required, or even useful, unless the optimization is implemented.)

5 Address Space Model

SAFE's primary protection mechanism is the imposition of a data object management scheme that employs a vector of **Address Spaces (ASs)**. An AS is identified by its associated **AS Tag (AST)**. At the highest level, SAFE's data objects are divided into four groups:

- **Transitional Data Objects:** data objects that have process-specific names that map to system-wide names (for example: ephemeral memory and file data objects). Such data objects have a single sys-tem-wide name (which in SAFE's AST model is a number) and potentially, multiple, independent, process-specific names (different and smaller numbers). Much of SAFE's security mechanism is asso-ciated with (1) restricting processes to their respective process-spe-cific name spaces: **Process Address Space Tags (PASTs)**; and (2) the mechanism employed to translate between a PAST and its associated system level data object: **System Address Space Tag (SAST).**
- **Public Data Objects:** which are encapsulated by an LLS. An LLS supports a public data object name (number) space represented by **Global Address Space Tags (GASTs)**. A GAST is the exclusive data object identification mechanism from outside a SAFE system. Each valid GAST identifies an object within an LLS. A SAFE implemen-tation can internally use a GAST in any number of ways. Within this description of an example SAFE system, GASTs are employed as external identifiers and **Backing Address Space Tags (BASTs)** are used as internal (private) data object identifiers (GASTs are 448-bit numbers and BASTs are more manageable 30-bit numbers). A GAST has no utility apart from its association with a BAST. GASTs are the only externally visible (public) artifact of SAFE's data object identification model.
- **Private:** The example SAFE LLS implementation described by this document employs a private address space supported by a vector of the Backing Address Spaces (BASs). A Backing Address Space Tag (BAST), the index of a BAS within an LLS, is the unique identifica-tion mechanism within the system for the management of persistent data storage. BASTs are manageable data object references (though large, the BAST space is small enough to enumerate and to be placed in a directly indexed table). BASTs are strictly internal to an LLS and never externally exposed. A BAST has no external visibility and is a completely private reference model (there is no mechanism via which a BAST can be directly manipulated, nor can it is used by a user space entity to identify a data object). BASTs are used to map data objects into their associated system level active data reference environment. [Note: BASTs are an implementation detail that represent an implementation option, not a required architectural component. However, BASTs have been codified within this document to simplify the associated description and to solidify the associated design complexity. BASTs also support the creation data spaces that are physically disconnected from external access.]
- **Managerial:** A system runs processes. A process is managed by its associated state, which is encapsulated by its **Context Address Space (CAS)** and identifying **Context Address Space Tag (CAST)**. A system, under control of its embedded Process Management Unit (PMU), uses this state to control the system's Processing Nodes (PNs). PN functionality and process management are tied together by CASTs. Within a system, processes communicate via a **Messaged Notification (MN) interrupt/signaling** mechanism and a pairwise process memory sharing model that, in concert, support **InterPro-cess Communication (IPC)**. IPC provides an extended functionality, which comes in the form of Notification Queues and a small-scale process-to-process data exchange mechanism, that unify both hard-ware interrupts and software signals within a single notification model.

There are five types of Transitional Data Objects within a system's 72-bit address space: (1) files and memory are unified and managed as lin-ear byte vectors; (2) network streams; (3) acceleration management (display/graphics being the dominant instance); (4) InterProcess Communication (IPC) interfaces which include: (4a) Messaged Notifications (MNs), (4b) **Shared Memory (SM)**, and (4c) **SAFE State Registers (SSRs)**, and (5) code modules.

ASs bridge the gap between system resources and process resources. ASs are used to enforce containerization via the construction of hardware-based ***Enclaves***[1]. A system has data. Programs are code that access that data. Programs run within enclaves. An Enclave has an asso-ciated Context Address Space (CAS) and it is defined thereby. A CAS contains vectors of available code, data, network, message, and graphi-cal objects. An Enclave's associated program (only one program can execute within an enclave) can access the Enclave's registered data objects (which reside in one of a CAS's object vectors), without resort of a system call. Objects that are not present in a program's CAS are functionally invisible to the program. All objects not explicitly named within a program's CAS are physically invisible (there is no mechanism via which the associated hardware can generate a reference).

An object outside a CAS, can be dynamically added via a system call. To add an object, a program must know the data object's name, and/or use a system based facility/capability that will search for a data object on the program's behalf. Data object identifiers are very large numbers (the 448-bit Key field of Object Identifiers (OAs)), the odds of guessing a valid OA are not functionally different from the odds of guessing a specific OA, both are functionally 0 (1.48^{-126} and $1.38*10^{-135}$ respectively). Hence, if a data object's identifier (name), its Global Address Space Tag (GAST), is not known by a program, then it is not accessible.

A GAST need not be known a priori by a program. An object can be mapped to a program by a system service, potentially without exposing the object's name to the program. In this case, the system, via the system owner's supplied configuration service module, will decide which data object(s) a program can potentially access, and then decide again whether a data object in the list of potentially accessible objects can be concurrently accessed within the program's current vector of accessible objects (a system can impose rules that preclude 2 objects from being concurrently accessible by: a single process, by an identified group of processes, or within the system as a whole).

Each AS instance is tagged with an OIT that uniquely identifies the associated data object throughout the associated system (an OIT's Key is used to identify the object, and its Value is used as a compressed object identifier, purely within the system, once the object is activated (made accessible to one or more processes)). An OIT Value becomes its the associated object's OIT Domain specific Address Space Tag (AST). Every associated instance (complete or fragmentary) of an object within a SAFE system carries its associated AST and is solely identified thereby. SAFE data objects are content addressed: the identity of any given datum is dependent on its (Type[2], AST, Offset), not its location in a physical store. An object's AST is associated with the object at every level in the system, from an Level-One Cache (L1) down to its at-rest storage within a system's Last-Level Store (LLS). The LLS that anchors this access model is a closed system and there is no mechanism for interrogating an LLS's internals[3].

A specific data object can be exclusively tied to a program or group of programs. The data object may not have an associated GAST, the data object is only identified by a BAST[4] tied to a **Data System Address Space Tag (DSAST)** and hence, from the perspective of all programs outside its inclusion group, the protected/private data object does not

1. Although the notion of *containers* is well understood, SAFE does not implement a container in the same manner as solutions like Docker (a trademark of Docker, Inc.). SAFE Enclaves execute on bare hardware. There is no underlying system that manages a containerization system overlay, from which an escape into the base Operating System can result in: "Cry havoc and let slip the dogs of war". SAFE employs the alternate term "enclave" to differentiate itself from the more prevalent and more prevalent notion of Docker-ish containers.

2. A data object's Type is not purely synonymous with its associated OIT's Domain. A Type can subsume multiple Domains. However, every Domain is a member of exactly zero or one Type (some Domains do not participate in a system's data/code 3-tuple access model (OIT, OAT, Computational Object), with the user Domain being an example of an (OIT, OAT) tuple).

exist. An accessor program can mediate all interaction with a data object by requiring all other programs employ IPC to request access services. These service interfaces can apply system specific identification mechanisms to determine which, if any, portions of an accessor interface can be invoked by any given program.

A SAFE process has an associated vector of process-specific ASTs. An allowed AS access is communicated to the system via **Process Reference (PR)**, which correspond to virtual addresses in a legacy system. PRs are nominally 64-bit wide, but the initial implementation of a standard size SAFE system only used the least significant 58 bits of a PR (the most significant 6 bits must all be clear/zero). A PR is translated from a process-specific reference to a system-global reference via a set of CAS-based tables that map a process' AST to the associated system's AST. A **System Reference (SR)** is a 72-bit address that identifies global resources (as opposed to a PR that identified process resources). There is an address range expansion model that allows for PR expansion into its otherwise reserved upper bits, and SR expansion into an expanded width format (more than 72 bits), when the default model's enormous range is insufficient (the default PR/SR model supports a million con-current processes; with 64 million concurrent network connections; a 50 million concurrently accessible data objects (files and memory allo-cations).

3. A direct LLS interface can be created, but this is an emergency recovery interface. Such an interface need not be enabled and the interface that unlocks an LLS can be made arbitrarily difficult to trigger, including the possibility of multiple, independent terabyte passwords (an LLS password can be made so large that the simple act of transmitting it creates a significant delay between the time that unlocking is begun and the earliest time it can possibly finish).
4. The private object can be preserved when its accessor program(s) are terminated by dynamically assigning it a GAST, and then disassociating (killing) the GAST when its accessor program(s) is reinitialized.

5.1 Private Address Space

Table 3: Backing Address Space Tag (BAST) Format

Bits 29:0
BAST

Table 4: Context Address Space Tag (CAST) Format

Bits 19:0
CAST

Backing Address Space (BAS) references are purely internal to a Last-Level Store (LLS) and are not a formal aspect of a SAFE design (*Note: the capability could be otherwise supported by an entirely different internal management mechanism). However, for purposes of this document, this implementation detail is elevated to the level of an architected SAFE component for illustrative purposes - much of what follows is easier to explain if this implementation artifact is treated as it is an architectural component.

Context Address Space (CAS) references identify user processes and SAFE **System Components (SCs)**. Any hardware that is capable enough to requisite a system service is identified by an SC Index and a CAST. A system controlled table contains the vector of SC to CAST translations.

A SAFE system's data is protected by externally inaccessible BASTs and a systems execution (processes) and I/O is protected by externally inaccessible CASTs. The exposure of both control mechanisms is via system controlled translation and assignment. Any aspect that does not employ a translation employs an internal reference check.

Persistent data is managed by an LLS, the terminating lower level of SAFE's Cache Hierarchy (CH) and the management engine of a SAFE system's data store. An LLS can theoretically abstract many forms of storage (in the short term they will primarily be rotational and/or networked (which in turn is a transitive abstraction of rotational storage and/or dynamically, demand constructed data content)) without ever exposing the nature of its backing storage model/technology/mechanism to the associated system.

An LLS uses Backing Address Space Tags (BASTs) as the handles (identifiers/names) for its associated data objects. Many of a system's BASTs are identified by OIT Value fields. A BAS is indexed by a 30-bit BAST (this billion data object capability can be restricted within smaller systems). BASs are the logical manifestation of an LLS-managed, physical, backing store. BASs are a convenient mechanism for establishing static data object sharing within a system. Multiple processes can each have their own independent system-level data object reference to a shared BAST value (a Global Address Space Tag (GAST), that is implemented as an OIT (KEY, Domain) tuple that refer-ences the indicated OIT's Value) .

There is no relationship between a BAST index value, a GAST index value, and any associated indices (process-specific or system-global) used to reference an active instance of the data object. A data object can be referenced by any BAST index and a BAST index can be referenced by any GAST index. Multiple independent GASTs can reference the same BAST. An LLS's closed, embedded system (hardware/software mash-up) abstracts the BAST management model (a BAST has no representation outside its LLS - its presence is only implied to individuals knowledgeable with regard to SAFE's internals). All aspects of GAST-to-BAST mappings and of DSAST-to-BAST mapping is managed exclusively by the associated LLS. The notion of a **BAST based DSAST-to-GAST** transitive mapping is an implementation detail (but one that is fairly essential to understanding of an interface that at the user level maps GASTs to DPASTs. An LLS can autonomously delete an unreferenced BAST.

A BAST index has no utility external to an LLS and because an LLS is a closed component, somehow discovering the index of a BAST that an LLS is using to manage one of its data objects has no objective utility. The LLS interface does not employ (ever expose) a BAST's value.

5.2 Public Address Space

Access to SAFE's data objects is controlled by an OIT/OAT interface whose Domain indices are embellished to include access permissions. A Global Address Space Tag (GAST) is composed of an OIT's 448-bit Key index and 32-bit Domain index. The combined 480-bit value is suf-ficiently large that guessing a GAST is computationally impossible (a GAST is not an encryption key for which correlated data exists - it is just a very big number). A GAST exists only as a mapping to a Backing Address Space Tag (BAST), which is the OIT's 32-bit Value field. These data/code access specific OITs are collectively referred to a GASTs [Note: A GAST/BAST is always synonymous with an OIT structure, a GAST is always the concatenation of an OIT's Key and Domain, and a BAST is always a OIT's Value.]

Data object protections are associated with a GAST, not its referenced BAST. A BAST is purely a data storage management structure. GASTs grant and manage access to BASTs. Multiple independent GASTs can concurrently access a common BAST with identical or radically differ-ent protection models. The GAST protection migrates to its referencing DPAST (protections are applied at the top of the SAFE Cache Hierar-chy (CH) via the characteristics of the GAST used to access the associ-ated data object).

Functionally a GAST never transcends its LLS (it is meaningless out-side of its associated LLS). Statistically, no two data objects on the planet will ever share the same GAST (GASTs are randomly assigned and there are a billion-billion-billion-billion-billion more GASTs than subatomic particles in the known universe, including all that theoretical dark matter stuff that supposedly represents a 5x multiplier on the known universe's mass).

A GAST references a BAST (SAFE's logical addressing model). BASTs represent SAFE's associated physical addressing model. BASTs are private and form the nexus of data management within a SAFE sys-tem. GASTs are public, but their utility is limited to their ability to iden-tify a BAST and the associated protections that they indicate are to be applied to their associated data object. A GAST is used in a software generated request to an LLS, but aside from the presence of a GAST in a system's open, create, and remap interfaces, they are useless. GASTs are not present in any SAFE hardware state outside the LLS (they have no presence/representation in CH or PN hardware).

5.3 LLS Internals: GAST, BAST, OIT, and OAT

Internally an LLS has **four** major management structures: 1) Object Identi-fication Tuple (OIT) Tree, 2) Object Authentication Tuple (OAT) Table, 3) BAST Table, and 4) Storage Management Tables. All four of the major management structures are critical to the operation of a system. The loss of any one of these structures will render a system's storage infrastruc-ture largely unusable. These structures all employ a persistent backing store and all of their associated storage blocks are to be encrypted (the data they reference may not be encrypted, but the management infra-structure used to access that data is encrypted). It is an essential practice to maintain multiple copies of a system's management infrastructure, preferably in multiple, physically diverse locations. If any one of these structures is lost so is all the data on the associated system. As indi-cated, all four are necessary. A geographically diverse back-up model based on the placement based on

5.3.1 OIT Tree

Every externally referable SAFE system object is identified by an OIT. All GASTs are OITs, but not all OITs are GASTs. An LLS contains a single unified OIT Tree that is indexed by (lookup uses) the targeted OIT's (Key, Domain) tuple. All GASTs managed by an LLS are present in its OIT Tree. An OIT Tree is LLS specific and a multi-LLS system expands a GAST lookup value to a 3-tuple: (LLS, Key, Domain)[1].

An OIT Tree is logically nothing more that a list of all the OITs in a system. An OIT lookup steps through the list comparing the externally known Key and Domain of to the Key and Domain of every OIT in the list until a match is found or the list is exhausted. Physically, such a list needs a sophisticated management structure to optimize system access (stepping through a billion OITs could take ten minutes). With 64-bytes per tuple, even a modest sized SAFE system's LLS cannot manage its OIT Tree exclusively in memory. A model that is backed by rotational and/or solid-state storage is required to minimize memory footprint and to ensure system resiliency (a memory structure would be lost if the associated system lost power and/or crashed. As a high level statement, **an OIT Tree has three implementation options**: Hash Table; B +Tree/B*Tree; or Balanced Binary-Tree (Red-Black Trees being the currently favored mechanism). OITs can be effectively cached.

A caching Hash Table that is backed by a B+Tree can be a highly effec-tive management scheme. The following is only provided as an imple-mentation aid and it not prescriptive:

- OITs are a very sparse address space that can easily be accommo-dated by a b+tree or b*tree.
- OITs are easily and usefully cached. Any substring (vector of bits) of a OIT's Key is a random hash of the OIT and could be employed for any Order-1 ($O(1)$), hash based lookup algorithm.
- A hash collision can easily be rehashed in the same hash table or a smaller, secondary hash table via the selection of a different sub-string from the OIT's Key (a substring adjacent to the prime sub-string is as random as any other).

OIT lookup is not a frequent activity and its latency is generally not a significant contributor to overall system performance. Conversely, fast and simple tends to be superior to slow and simple and given an OIT Trees dependence on relatively power consumptive storage hardware, a modicum of algorithmic sophistication to minimizes usage of a system's resources is warranted.

An OIT Tree is the corollary to a file system in a legacy OS infrastructure. For all of SAFE's debt to Multics, SAFE does not use the only sig-nificant enduring feature of Multics: a hierarchical file system. The **OIT Tree is a flat structure**. There is no hierarchy and there is no associated transitive reference model (an object is never relative to another object) SAFE supports the construction of virtually hierarchical **File System Views (FSVs)**, but this is a software managed artifact that is not reflected in the underlying system. A system can support many FSVs (potentially millions). Each FSV is a data object referenced by an OIT,

1. An LLS does not use a LLS field, but an LLS field is required to steer an access request to the appropriate LLS.

whose Value is an index into the associated LLS' OAT Table for encryption and authentication, and by a BAST Table for storage management.

An LLS's OIT Tree is completely encapsulated by its LLS. It is not exposed to external inspection. All aspects of its external manipulation are strictly managed by messaged requests. No processing or memory resources are shared by a entity that can request an OIT dependent service and the LLS that supplies the service..

5.3.2 OAT Table

An OAT Table contains a list of all of a system's automatically managed encryption keys and authentication checksums. An OAT is not a searchable value. An OAT is identified by an OIT Value field. An OAT Table is indexed by a Value and the associated OAT is extracted. OAT Table indices (OIT Values) are not necessarily a sparse mapping and an OAT Table can be directly indexed (this is true for even relatively large tables). An OAT has persistence requirements and even if OAT Table's size makes memory a reasonable management local, a shadow of a sys-tem's OAT Table(s) needs to be maintained in a persistent storage node. OATs are frequently accessed and a hashed, cached memory structure for an SSD or disk based table is a reasonable model for OAT Table management.

OATs are an integral component in SAFE's encryption model. The OAT Table itself is encrypted. This creates a catch-22 at system initialization (you need to be able to decrypt to read that table that has the decryption tokens). Access to the OAT Table requires that an LLS be capable of storing and secreting a single OAT outside the OAT Table. If the major LLS management structures rely on well known storage locations and table indices, then the only aspect of system initialization that is not defined by default is the decryption key and checksum of the OAT Table.

5.3.3 GASTs

As previously noted, all GASTs are OITs, but not all OITs are GASTs. The partitioning of a system's OIT vector into GASTs and non-GASTs can be accomplished via the OIT's Domain field. A set of GAST domains is exclusively associated with the set of OITs that reference BASTs. All OITs with one of the BAST specific Domain indices reference BASTs and no BAST is referenced by a OIT with a non-BAST specific Domain index. GASTs reference BASTs (OITs contain Values), but not all BASTs are referenced by GASTs. A BAST lookup employs what can be considered a partial OIT (Key and Domain, but no Value). A BAST lookup returns the matched OIT's Value field (BAST) or 0 (teh null BAST). There are a computationally infinite number of OITs and knowing an OIT is considered equivalent to having the name and password of the associated data object).

5.3.4 BAST Table

A BAST Table contains a list of all the of the associated system's BASTs. BAST Table element identifies a small set of size-dependent, LLS storage hierarchy entry points (where the data is located) and addi-tional metadata that indicated how the data can be accessed and modi-fied (another set of these fields are in a GAST, represented by characteristics encoded within a Domain field, but the BAST versions act as overrides that supersede any less restrictive GAST supplied char-acteristics). As with the OAT Table, a BAST Table can be directly indexed, cached, requires that its persistent image be maintained. BASTs and OATs are largely peer/parallel structures, though there are: OITs that have OATs, but no backing store (no BAST, the OIT Value must be considered to be an OAT Index); OITs that reference BASTs that are not encrypted (no OAT)[1], there are BASTs that have no refer-

1. Although encryption is generally applied to all data objects, there is no encryption requirement. Encryption is a moderately power intensive action and if a copy-write permitted instance for the text of *The Cat in the Hat* is being stored on a system, it is reasonable to be able to forgo its encryption.

encing OIT and potentially no associated OAT, but there are no OATs without an affiliated OIT or BAST (there must be something for the OAT to encrypt).

The co-location of an OAT with its associated BAST is an option (they are generally both associated with a data object). However, the separa-tion of these essential structural components into independently encrypted and protected structures makes it harder to compromise a system.

5.3.5 Storage Management Tables

An LLS relies on multiple sets of size differentiated data vector management tables. SAFE uses relatively large storage blocks to create relatively small, per data object block counts (relative to armies the 4kB storage blocks required to store data on legacy systems. The details of a sample LLS storage management implementation are presented in the SAFE Reference Implementation Manual. The only salient characteristic of SAFE's LLS storage management infrastructure is that, aside from its free lists, they are exclusively referenced by BAST Table Entries.

5.4 PMU Internals: CAST and SCI

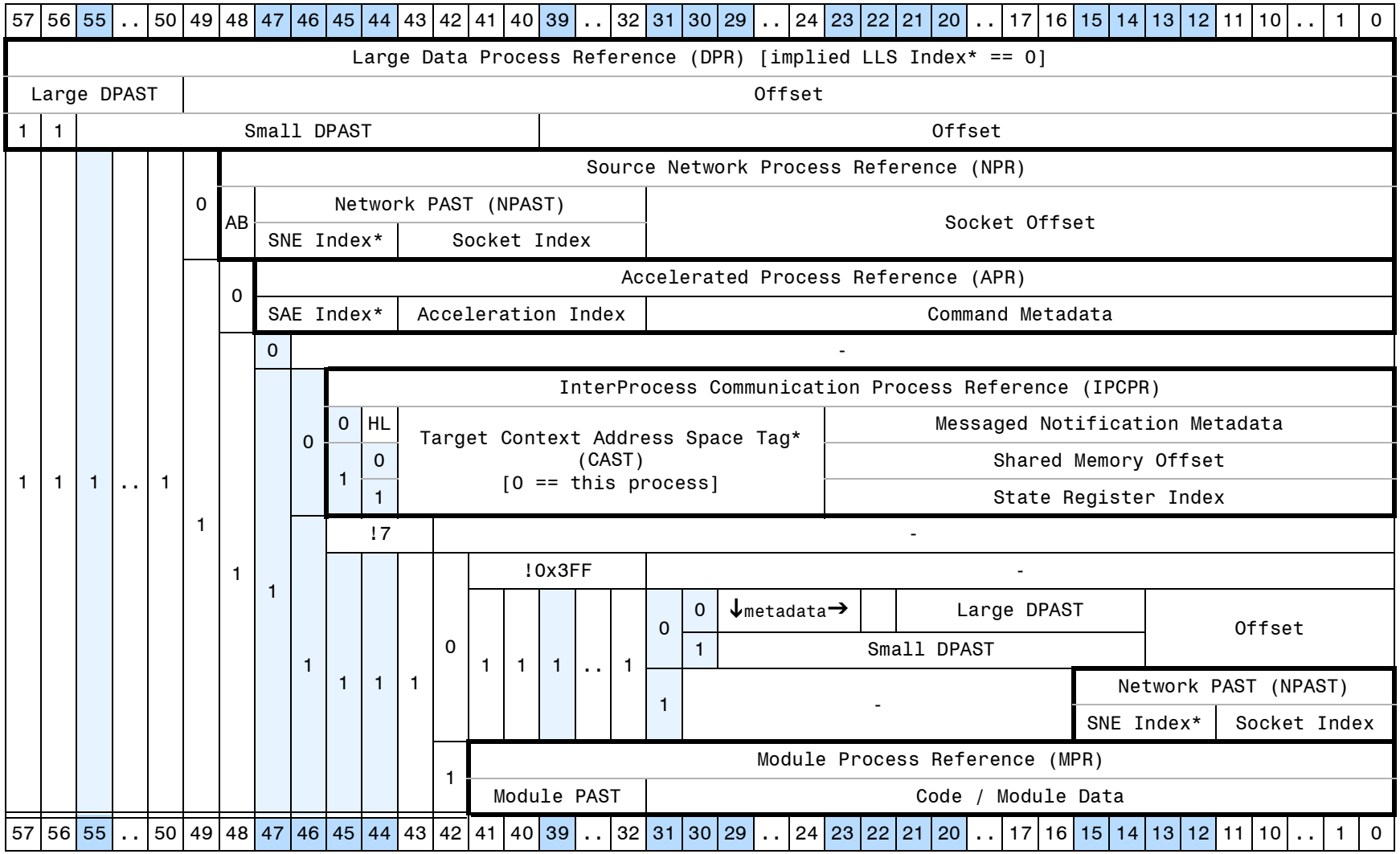
Internally a PMU runs off a single overarching control structure: a CAS Vector. A CAS controls process execution and a CAS Vector is the list of all the currently executing processes. There are multiple ancillary virtual control structures that are embedded within the CASs of the respective processes. A CAS Vector is too large to fit in a PMU and a PMU owned Data Address Space (DAS), based in the LLS associated with a PMU, is used as the backing stote for the CAS Vector. A PMU caches the relevant data from its CAS Vector. A PMU controls process execution and can therefore marshal the requisite process state before switching to a new process. A PMU has dedicated processing resources and queuing future process state is easily accomplished. Next genera-tion PNs could add a hardware thread that would allow a PMU to use a quiesced hardware thread for background context switching.

A CAS Vector is a linear array of 16mB, per process CASs. The vector ios the exclusive purview of its associated PMU. The DAS in which the CAS Vector resides exists within an excluded DAS range (only a PMU with a matching index can generate the DAS and an inherent Protection Exception is generated if the DAS appears anywhere except between its PMU and LLS).

SAFE SCs do not rely on translation to generate System Address Space Tags. A secondary security mechanism is layered on the set of hardware generated address references to ensure correct operation. Every SC has an associated CAST. A PMU contains a table or a virtual table. A default SAFE implementation will employ a virtual table in which a system can have a maximum of 511 non-SAE, non-SNE SCs. Higher level cache components are not CAST indexable (they rely of cache protocol targeting) and hence the number of SCs that are neither SNEs not SAEs is generally very small. A moderate size system will have at most a couple LLSs and a similar PMU count, which means that a multi-thousand application processor core system can be suported with a virtual table. A physical table will have perhaps 4096 entries of 20 bits each that identify for a given System Component Index (SCI), its CAST. The virtual table’s SCI and CAST are the same.

5.5 Process Address Space Formats

Table 5: Process Reference Formats ('*' denotes field that expands into otherwise zeroed bits 63:58 when expanding address range)



SAFE's standard **Process Reference (PR)** is 58 bits wide and is generally sufficient to the needs of moderate to large systems. The table above is explained in significant detail later in this document. For now, SAFE partitions access in to a set to type specific access ranges and each of the ranges is associated with a class of System Component (SC) that manages the associated data type and a specific instance of that class that manages the referenced object.

An SC abstracts some aspect of canonical Operating System (OS) behavior in a way that allows for service requests and management to transit a Cache Hierarchy (CH) that physically connects processes to SCs. The use of SCs as replacement for OS software services obviates the need for a Processing Node (PN) that runs user processes to ever transition to a privileged state/mode. User processors only run user pro-cesses. [**Note**: There is a mode in which a process is allowed to send system level notifications, but this is tied to a process' Context Address Space (CAS) and not mutable by a process (all CASs are physically invisible to all user processes - the CAS access mechanism in the SR below is not replicated in the PR above).]

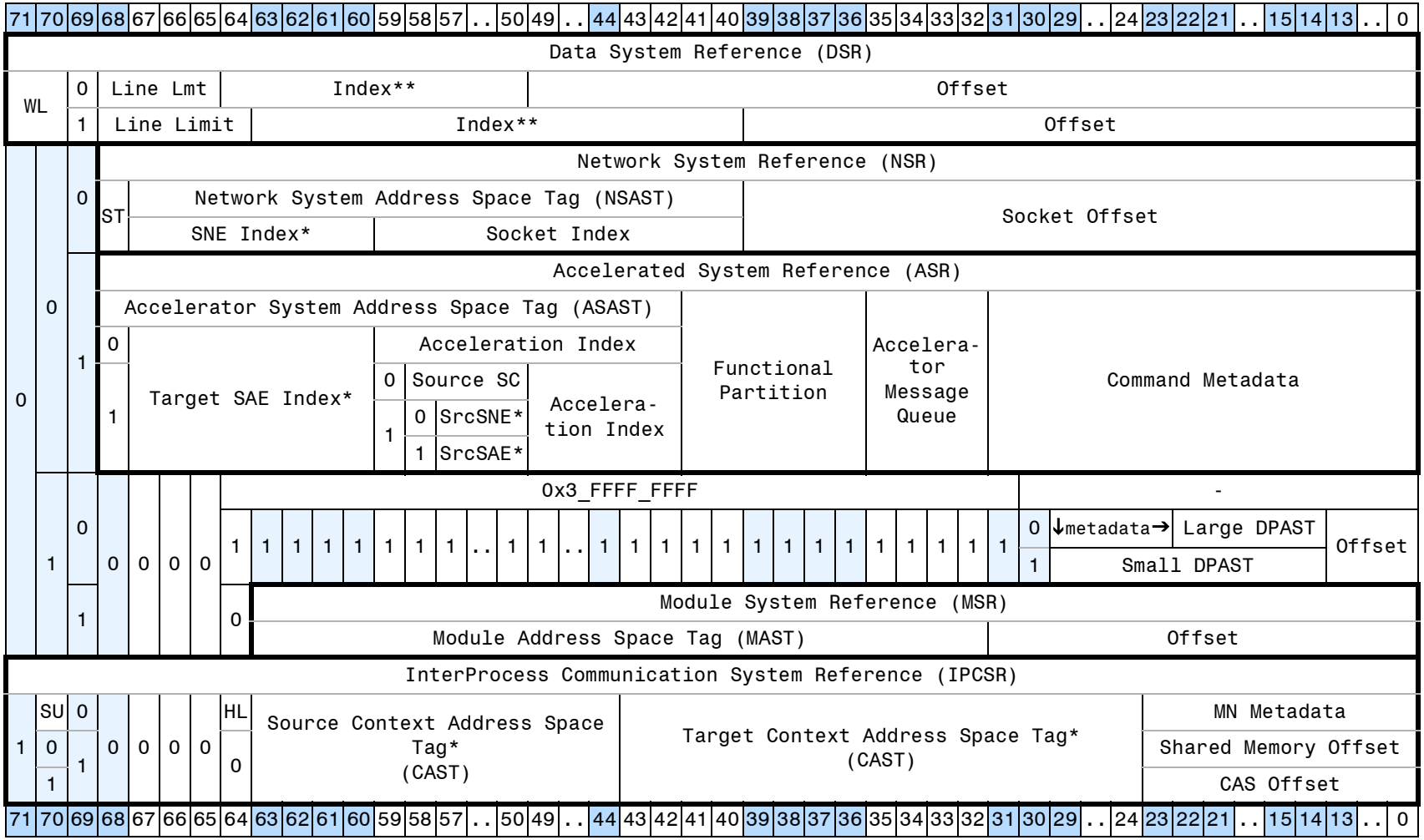
SAFE supports a bolstered system capability model in which the widths of specific fields are expanded via the progressive allocation of bits 58 through 63, Each bit of PR expansion requires that the associated System Reference (SR) be expanded by 2 bits (PRs only specify Target CASTs, with the source being implied, while SR must enumerate both source and target, hence and additional expansion bit is required for the implied source). A fully expanded (64-bit) PR and will support an enormous system (a zebibyte of storage and 100's of terabits of networking). PR expansion fields are marked with a '*' in the enumerated formats above. In practical terms, the risk of localizing/concentrating a resource set that transcends the capabilities of the default PR format probably exceeds the value of doing so.

Reserved address encodings offer an ability in add additional capabilities to a SAFE system, For example, a 16mB Private Local Store[1] capability could be added via an allocation of address in which bits 63:24 are equal to 0x3_FFFB_FFFF.

1. A **Private Local Store (PLS)** is a small, untranslated, caching hierarchy independent memory associated with a CPU. It is a directly addressable RAM that has no coherency nor caching capability. It generally holds process specific data and is loaded and stored (like registers) on a per context switch basis. However, options exist to make it a very fast shared memory for processes/threads that have tightly defined/controlled execution sequencing. That is, if a PMU were configured to, for example, sequester a processing core on which it swap threads X and Y, and to not swap-out the PLS in association with a thread switch, then the sequestered core's PLS, could function as a very high performance data sharing mechanism. [Note: A PLS and the proposed enter-thread/process PLS sharing capability are easily supported by SAFE's infrastructure, but are also fundamentally orthogonal to SAFE.

5.6 System Address Space Formats

Table 6: System Reference Format



MN = Messaged Notification **SAE** = Secure Accelerator Engine **SNE** = Secure Network Engine **SU** = System(0)/User(1)

CAS = Context Address Space **SC** = System Component **Src** = Source

HL = High(0)/Low(1) Priority **SM** = Shared Memory **ST** = Source(0)/Target(1)

SAFE's standard/default System Reference (SR) is 72 bits wide. An SR is the global reflection of the translation of a processes PRs into the sys-tem's global address space. SRs can directly access a restricted range of SC (those associated with its data type and specific associated object instances. User processes can only indirectly, via PR-to-SR translation, access n system resource (an object identified by an SR). A PN's SAFE adaptation layer contains a vector of process context isolated translation resources (not altogether different from a legacy CPU's Translation-Look-aside Buffer (TLB)). These translations make it possible for a process to safely access global resources via tables that are tightly con-trolled by the associated SCs and inaccessible to user processes (the inaccessibility is absolute, there is no hardware model that supports a user process' access to a process resource translation).

5.7 Data Address Space (DAS)

A **Data Address Space (DAS)** is a container for a linear vector of bytes. The bytes may be logically grouped in ways that make their granularity logically other than bytes (for example a Unicode file or a IEEE-754 Floating-Point vector), though at the physical level the system treats them as bytes. From a legacy perspective, data is partitioned between potentially persistent data allocations (files) and ephemeral data alloca-tions (memory). Functionally, a file is a randomly indexable, readable, writable, linear sequence of bytes. Functionally, memory is a randomly indexable, readable, writable, linear sequence of bytes. As Multics noted more than 50 years ago, files and memory are the organization-

ally identical, and they can be cached within a common infrastructure[1].

SAFE unifies memory and files into a completely undifferentiated infrastructure based on DASs. There are 3 connected forms of DASs:

- Backing Address Space (BAS): referenced by a Backing Address Space Tag (BAST) - managed by an LLS - represents an LLS's per-sistent data object infrastructure - this is a system global infrastruc-ture component.
- **Data System Address Space (DSAS)**: referenced by a **Data System Address Space Tag (DSAST)** - mapped to a BAST - isolated to a Cache Hierarchy (CH) - represents an indirect BAS access mode -this is a system global infrastructure component.
- **Data Process Address Space (DPAS)**: referenced by a **Data Process Address Space Tag (DPAST)** - mapped to a DSAST - isolated to a PN's **SAFE Adaptation Layer (SAL)** - represents an explicit, poten-tially operationally restricted, DSAST access permission (which is generally a transitive BAS access permission) - this is a process spe-cific infrastructure component that can be indirectly exercised by a process' DPAS-to-DSAS translation mechanism, but the translation itself is not directly accessible by the associated process.
- **Module Data (MD)** Configuration-based Access: Modules[2] can have up to 4 Module specific BASTs. these BASTs reference private/static Module accessible data, that is not visible and/or accessible by any other Modules in an associated program. A Module has an MD configuration that indicates the state of its MD BASTs and automat-ically injects a call to the Module's initialization function if a Null MD is encountered when accessing an MD BAST that is configured for use. Large DPAST index 191 (the last large index) and Small DPAST indices 65_532:65_535 (the last 4 small indices) are dynam-ically mapped to MD BASTs.

A DAS lives within a Last Level Store (LLS) and is cached into a SAFE system through a Cache Hierarchy (CH). The DAS itself never leaves the LLS and no aspect of its management is ever surrendered to any other management entity within a system. DASs are represented with a system via the caching of their constituent pages. A DAS is typically tied to a BAS, the system's persistent data management entity. Inactive DASs (those not currently accessible by any processes) are persistently mapped to BASTs. A system's active data object management system uses DSAST to identify its active BAST. There is no relationship between a BAST and any given DSAST to which it is mapped. After a DSAST is retired (the associated BAST is no longer accessible via the DSAST), any of a BAST's future DSAST mappings is very unlikely to involve any of the set of its recent DSAST assignments. A BAS is made accessible by mapping its BAST to a DSAST and then giving a process access to the DSAST by mapping one of the process; Data Process Address Space Tags (DPASTs) to the DSAST.

A process generates a **Data Process Reference (DPR).** DPRs are trans-lated into **Data System References (DSRs)** via the associated process'

1. It is interesting to note that in the wake of Multics, files and memory remained completely independent resources managed by wholly separate OS kernel mechanisms. This in spite of an ever increasing blurring of the notion of file persistence and memory impermanence. The management of virtual machine instances being an excellent example.

2. Modules are code based structures that can hold code, constants, and entry points (a callable vector of code offsets that represent the only mech-anism via which a module can be entered).

Context Address Space (CAS)-based DPAST→DSAST Translation Table (DPAST→DSAST, the use of the '→' character is exclusively associated with the CAS-based translation table and its presence implies Translation Table). A process' CAS has a fully enumerated DPAST→DSAST, but a CAS is too large to hold in a PN's SAFE adap-tation layer. The DPAST→DSAST is cached in the associated execut-ing PN's DPAST-to-DSAST Translation Table (`DP->DS_T`) and DPAST-to-DSAST Translation Cache (`DP->DS_C`). The `DP->DS_T` holds/translates specific DPAST indices and lacks the flexibility of a `DP->DS_C`, in which a smaller number of cache entries can support a much larger set of CAS-based translations. Conversely, the table is sim-pler, faster, and lower power than the cache; and a small set of DPASTs can be mapped to almost universally required set of DASs (for example, program constants, stack and heap).

DAS growth is managed by an LLS via the allocation of L5 cache lines. Shrinking a DAS is handled via Messaged Notifications from a process to an LLS. An LLS shrinks a DAS by first checking to make sure that the shrink is coherent with any other potential DAS users. An LLS shrinks a DSAST via a series of cache line invalidations.

A process can concurrently access 192 **Large Data Objects (LDOs)** and 64_512 **Small Data Objects (SDOs)**. An process LDO uses a range restricted (its 2 most significant bits cannot both be set), 8-bit DPAST to access a 50-bit Offset, which allows an LDO reference a 1pB data object. An SDO uses a range restricted (its 6 most significant bits can-not all be set), 16-bit DPAST to access a 40-bit Offset, which allows an SDO reference a 1tB data object. A process' CAS has a 64_704-entry (64_512+192) table in which each entry contains a 32-bit DSAST index and the associated process' access permissions relative to the refer-enced DSAST. The DPAST→DSAST is indexed by DPASTs. The data returned from the table defines the process; data horizon. Data in the table can be accessed with the associated permissions, Data not in the table is hardware inaccessible (there is no mechanism that allows a pro-cess to form an address that is outside the data horizon defined by the contents of its DPAST→DSAST).

A SAFE system minimally[1] supports 32_768 concurrently accessible LDOs and 16*1024[2] SDOs. A system employs Way Limits and Line Limits, in addition to an Index. The combination of fields from which a DSAST is constructed support relatively large DAS pools that reference data objects with size characteristics identical to those of a process' DPASTs.

No process can interact directly with a system resource (all resources belong to the associated system and a process is assigned a mapping to a system resource, the process can only use the system resources to which it has a mapping). All process specific references are translated to system references via DPR-to-SSR translation, of which the DPAST→DSAST is one. DPASTs exist solely within a PN's and trans-lation of DPASTs-to-DSASTs occurs at the PN/L1 Cache boundary (System Components (SCs) can directly access system resources). An implied LLS Index field served as the expansion field for this PR com-ponent (the LLS Index of the default PR is 0, `LLS[0]`). If the PR is expanded, the LLS Index DPAST boundary is flexible, with a process' DPAST-to-DSAST translation capable of referencing multiple LLSs and/or of expanding the range of an LLS's DPASTs.

At any given point in time, the potentially vast array of data objects (BASs) that are not mapped to DSASTs are inaccessible to any/all of a system's programs. A program cannot independently map a BAS to a DSAST unless it knows the BAS's associated GAST value (or one of its associated GASTs), and even then it can only request a mapping from its PMU.

In addition to standard data objects, SAFE supports a vector of Module private data objects. A Module DAS cannot be accessed outside its defining Module (its visibility is limited to functions within its Mod-ule). SAFE employs a shared Module model in which multiple pro-

1. A 2-bit Way Limit and a 5-bit Line Limit can extend a DSAST's uniqueness by a factor of 93 (3 valid Way Limits * 31 valid Line Limits), but in practice the number will be much smaller, perhaps a small as 7 or 8.

grams can concurrently share Modules and each shared Module can have program instance specific, but Module restricted; and/or globally shared DASs, in which every program/process that uses the Module shares a single instance of the Module's private DAS. The single DAS instance model (all processes that share the Module share a single instance of one to four DASs) represents a form of software mediated shared memory.

The following is non-standard relative to legacy systems and will only be an active data management mechanism in future SAFE aware systems. Programs can be constructed that employ Modules that overlap with those used by other programs. Overlap can be complete, partial, or completely disjoint. A program can use shared Modules and private Modules. A SAFE system can have a bit more than 32 million shared Modules and the same number of private Modules. Private Modules have no shared DAS capability, but they can employ program instance specific protected/static DASs. A SAFE system's shared Modules have an associated $2*1024^2$ by 4 entry DAS Translation Table. The table is the mechanism used to maintain Module shared DAS coherency. In general, a system will position Modules (assign indices) that employ shared DASs into blocks that minimize the allocation of backing store required for the system's Module DAS Table.

A SAFE Module can reference 1023 additional Module relative Module references. A SAFE program can reference a maximum of 65_536 Modules. A program manages its program specific Module DASs via a program instance (process) based Module DAS Table. When assigning program-specific Modules indices to global Modules, they will typically be ordered with Modules that employ Module DAS first, to minimize the amount of the associated process' CAS that will require backing store.

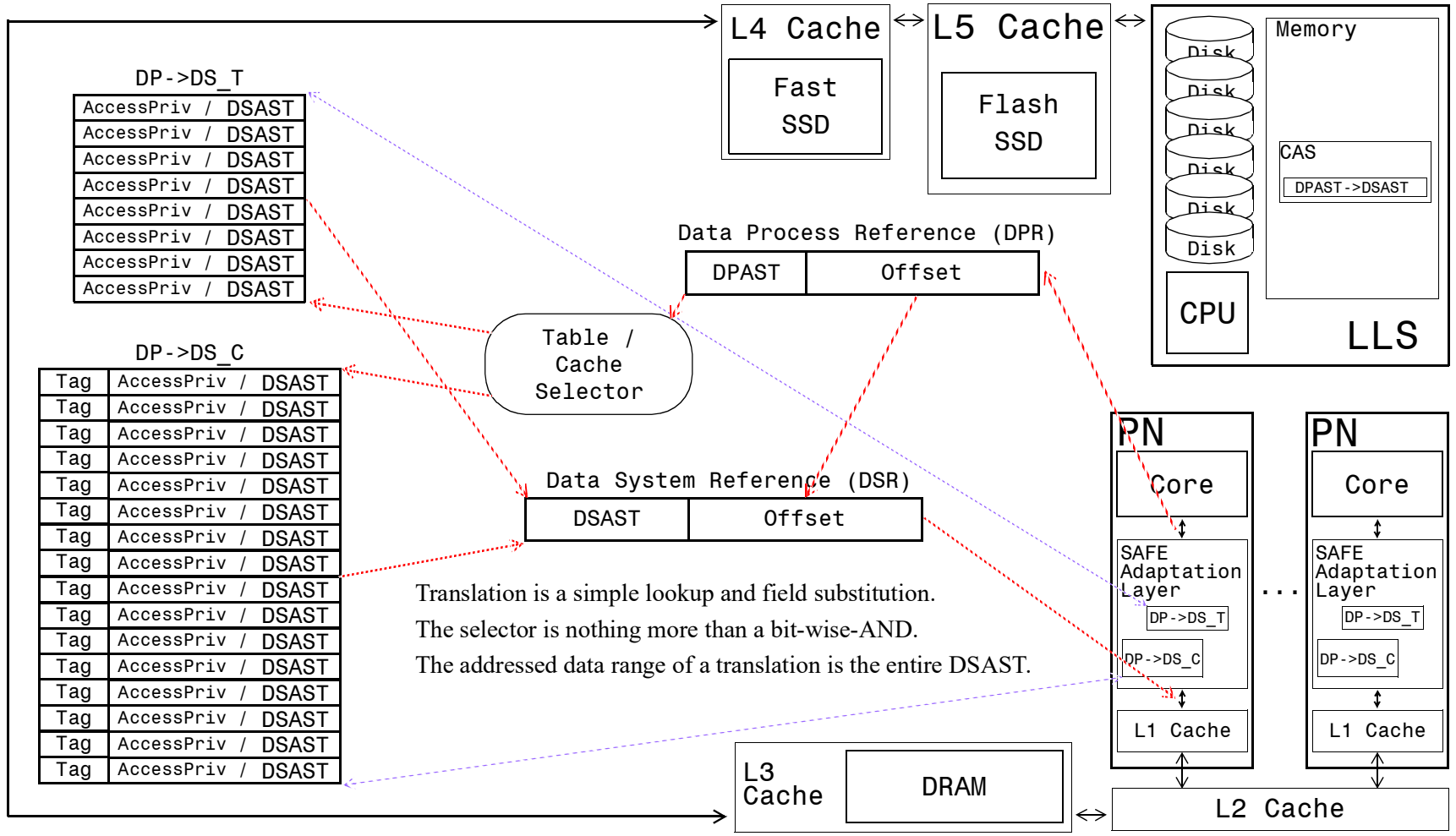
A PN's Adaptation Layer contains a 4 element PN Module DAS Table and the Module index of its current contents. During execution, whenever a program accesses `LDPAST[188]` through `LDPAST[191]`, the state of the executing module is checked for its Active Module DAS Count (LDPASTs are allocated in reverse order, that is, a count of 1 indicates that `LDPAST[191]` has an associated Module DAS and that all the other LPDASTs have their normal associations). If a count check indicates that a Module DAS is being accessed, then the current Module index is compared to that of the PN Module, and if they do not match the PN's current Module DAS Table is cleared, the PN's current Module DAS Module index is set to that of the current Module, and the accessed DAS entry is load from the System Module DAS Table or Program Module DAS Table indicated by the System Module DAS Count)[1]. A Module with Module DAS capability short-circuits the associated process' DPAST-to-DSAST translation for DPASTs in the Module DAS region (`LDPAST[188]` through `LDPAST[191]`). The short-circuit operation does not invalidate the affected translations LDPAST-to-LDSAST, it just makes them inaccessible when inside a Module configured for Module DAS usage.

A Module DAS can be associated with a persistent or ephemeral DAS. The DAS can require initialization via an program that uses an OIT and OAT to load an MD Table via an LLS's GAST tree; or is can be dynamically created the first time the object is used/touched, which also requires that a BAST be loaded into a table. A Module has complete (within its scope of influence) and independent control over the identity and content of its associated Module DASs.

1. System Module DASs are placed in higher numbered indices. For any pair of Module DAS counts the Module DAS configuration of the associated Module is trivially determined:

```
sys_mod_das = prog_mod_das = 0
if      (index >= sys_das_count) sys_mod_das  = 1
else if (index >= mod_das_count) prog_mod_das = 1
```

Figure 1: DPAST to DSAST Translation



5.7.1 Data System References

Table 7: Data System Reference (DSR) Format

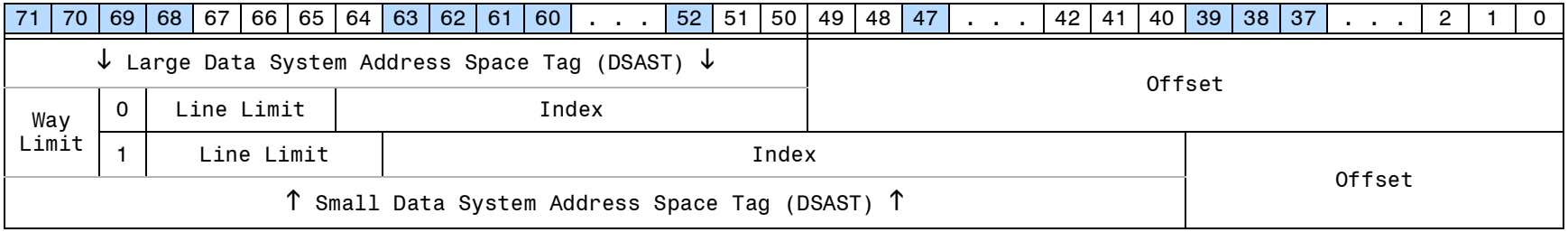


Table 8: Data System Reference (DSR) Field Descriptions

Bit Range	Field Name	Field Description
39:0 / 49:0	Offset	This field contains the Offset within the indicated DSAST of the associated reference. A Offset is 40 or 50 bits in size, depending on the size model of the associated data object (controlled by DSR bit 63). A large (50-bit) offset does not imply that the associated data object requires the larger offset's addressing range.
71:40 / 71:50	Data System Address Space Tag (DSAST)	Small data objects have a 32-bit DSAST and large data objects have a 22-bit DSAST. In both cases the DSAST contains 4 sub-fields, that in concert, represent the DSAST. The variable sized DSAST value is its associated data object's system-wide identifier. Almost 30% of the values within the DSAST's ranges are not valid identifiers (the invalid DSAST values are reserved or used for alternate DSR encodings). Two of the sub-fields impact the associated data object's cache placement, the third is a flag indicates whether the DSR references an LDO or an SDO, and the fourth is purely an identifying index, however, even the cache placement limiters are useful data object identifiers.
63:40 / 63:50	DSAST: Index	This field contains a 24-bit or 15-bit index value, that in concert with the cache placement restrictions (Line Limit and Way Limit) constitutes a 33-bit or 23-bit DSAST. Most DSASTs are usefully restricted either as a statement regarding their size or their relative importance. This field, independent of cache line placement restrictions, supports a concurrent, system-wide minimum (cache line placement restrictions act as Index multipliers) of $16*1024^2$ small DSASTs (1tB Offset range) and 32*1024 large DSASTs (1pB Offset range).

Table 8: Data System Reference (DSR) Field Descriptions

Bit Range	Field Name	Field Description
(LDO) 68:65 / (SDO) 68:64	DSAST: Line Limit	Cache placement restrictions can be applied to the cache line range into which a DSR can be placed. For example, a 1024-line cache could, for a certain set of data objects, be limited to its third block of 64 cache lines, using only 6 bits of data address for line selection within the restricted block's range. Conversely, a data object could be limited to the first half of a cache and another data object could at the same time be limited to the second half of the cache; both of cache lines could use the same Index value because the two data objects will never collide with each other within a cache. Within SAFE, this field restricts the associated data object to a range of accessible cache lines. Each cache is positionable into 16 equal size partitions, named: `p0` through `p15`. A DSAST is assigned to an aligned power-of-2 sized partition region that can be as small as a single partition (any one of the possible 16) or as large as the entire cache (the unrestricted partition - all 16 partitions are available). The LDO interpretation of this field is identical to the SDO's interpretation, with the exception that an implied 0 for, the least significant bit of this field (what in the SDO is bit 64), is assigned to LDO decodings (an LDO's most restrictive encoding limits an LDO to one-eighth of the cache instead of the one-16th that the SDO encodings support). The least significant set-bit of the this field determines the partition size and the remaining upper bits of the field determine to which of the possible ranges, of the indicated size, the DSAST is restricted. If field bit 64 is set, then the remaining 4 bits (68:65) index a 1/16th size partition. If field bit 64 is clear and bit 65 is set, then bits 68:66 indicate the location of an even index aligned partition pair (partitions 0:1, 2:3, 4:5 „, or 14:15). If field bits 64 and 65 are both clear and bit 66 is set, then bits 68:67 indicate the location of a partition quad (partitions 0:3, 4:7, 8:11 or 12:15). If field bits 64 through 66 are all clear and bit 67 is set, then bit 68 indicates that associated data object is restricted to either the first half (partitions 1:7) or the last half (partitions 8:15) of the of a cache. A field value of 16 indicates that the associated data object enjoys unrestricted cache placement. The field value 0 is reserved. Partitioning is not applied to the L1. Other cache levels can apply partitioning as appropriate to their respective sizes. Two DSASTs that differ in their Line Limit assignments are fundamentally different DSASTs, even if they have identical Way Limits and Indices; and even if the associated cache placement restriction delta is not applicable to the cache in which they reside. A BAST does not have a Line Limit. Access restrictions are applied as an artifact of DSAST mapping. A data object that has been mapped to an DSAST cannot change its Line Limit (the existing DSAST must be retired and the data object must be mapped to a new DSAST to modify any of a data objects active access characteristics). All processes that share a BAST/DSAST must employ identical restrictions (placement restrictions are components of a DSAST and hence the only real configuration requirement is that the system use the same DSAST for any shared BAST). If a system employs a compressed DSAST, it is the least significant compressed bits become default 0s and the remaining bits have their standard interpretation (an LDO is functionally employs a compressed SDO Line Limit).

Table 8: Data System Reference (DSR) Field Descriptions

Bit Range	Field Name	Field Description
69	DSAST: Size	This flag, when clear, indicates that the associated data object employs a small DSAST encoding (33-bit DSAST Index and 40-bit Offset) and that its access range is limited to the referenced data object's first tebibyte (the referenced object can be larger than 1tB, but any bytes that fall beyond its first 1tB are inaccessible via the SDO mapping). An enhanced SAFE DPAST→DSAST management mechanism supports partial/windowed/bounded DAS access via a DPAST (a window, and/or multiple concurrent windows that are shared by a single process and/or multiple independent processes, can be created in which a DPAST→DSAST has an associated base address and size and access occurs on a relative basis within the configured window boundary). When set, this flag indicates that the referenced data object can be as large as 1 pebibyte in size and that it employs a large DSAST encoding (23-bit DSAST Index and 50-bit Offset).
71:70	DSAST: Way Limit	Cache placement restrictions can be applied to the number of sets into which a DAS can be placed within a set-associative cache. For example, a 4-way set-associative cache could, for a certain set of data objects be placement limited to a pair of sets, making an associated cache, for that data object, logically a 2-way set-associative cache (only the first 2 or the last 2 sets are searched for the associated data object). Within SAFE, a DAS can be cache-way unrestricted (all ways are accessible), a DAS can be only be placed in the the first half of a cache's ways, or a DAS can be only be placed in the last half of a cache's ways. A DAS cannot be restricted to a single way and a cache and 2-way and direct mapped caches ignore a DSR's Way Limit (they the field maintains its role in DSAST differentiation). Field values are interpreted as follows: **0** = RESERVED; **1** = no way restriction (all ways available); **2** = only the first half of a cache's ways are available (e.g. in an 8-way cache, the associated DAS is restricted to cache ways 0:3); **3** = only the second half of the ways are available (e.g. in a 16-way cache, only ways 8 through 15 are available). Way restriction has no effect on fully associative caches. Two DSASTs that differ in their Way Limit assignments are fundamentally different DSASTs, even if they have an identical Size, Line Limit, and Index. A BAST does not have a Way Limit. Way limiting is applied as an artifact of a BAST to DSAST mapping. A data object that has been mapped to an DSAST cannot change its Way Limit (the existing DSAST must be retired and the data object must be mapped to a new DSAST to modify any of a data objects active access characteristics[a]).

a. The standard SAFE specification does not mandate a more sophisticated implementation of limited DSAST remapping.An implementation may optionally choose a more sophisticated limit modification model. The standard model is overly restrictive, as is the associated Line Limit model. Both limits could support inclusive limit relaxation. For example, a way limited DSAST could become a way unlimited DSAST if all instances of the associated DSAST's translation mappings can functionally be changed at the same time. Similarly, for example, a DSAST that is limited to a single sixteenth of a cache could be expanded to use an eighth, quarter, or half of a cache, as long as the relaxed restriction is inclusive of the DSAST's previous restriction (for example a restriction to `p3` could become a restriction to `p2:p3`, `p0:p3`, `p0:p8`, or `p0:p15`). Transitioning to an nrestricted placement is inherently inclusive and hence always possible. Another implementation option is to only support restriction relaxation for unshared DSASTs.

5.7.2 Data System References

Table 9: Data Process Reference (DPR) Format

57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	. . .	2	1	0
Large DPAST								Offset																
1	1	Small Data Process Address Space Tag (DPAST)																Offset						

Table 10: Data Process Reference (DPR) Field Descriptions

Bit Range	Field Name	Field Description
39:0 / 49:0	Offset	This field contains a 40-bit Offset (when bits 57 and 58 are both set) or 50-bit Offset (when either or both of bits 57 and 58 are clear) within the associated DPR. This field is copied directly into the DSR generated by the associated reference.
(LDPAST) 57:50	Large Data Process Address Space Tag (LDPAST)	This field contains an 8-bit Large (50-bit Offset) Data Process Address Space Tag (LDPAST) that can reference a data objects as large as 1pB. Large DPASTs are restricted to a range of 0:191 (192 references), the upper bit pair cannot both be set. This is the index in the associated PN's (the PN executing the associated process) DPAST-to-DSAST Translation Table, or the index used for its DPAST-to-DSAST Translation Cache lookup. The translated DSAST is used to generate a DPR.
(SDPAST) 55:40	Small Data Process Address Space Tag (SDPAST)	This field contains a 16-bit Small (40-bit Offset) Data Process Address Space Tag (SDPAST) that can reference a data objects as large as 1tB. Small DPASTs are restricted to a range of 0:64_511 (64_512 references), the upper six SDPAST bits cannot all be set. This is the index in the associated PN's (the PN executing the associated process) DPAST-to-DSAST Translation Table, or the index used for its DPAST-to-DSAST Translation Cache lookup. The translated DSAST is used to generate a DPR.

5.7.3 CAS Based DPAST→DSAST Translation Entry

Table 11: DPAST→DSAST Entry Format

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="22">↓ Large Data System Address Space Tag (DSAST) ↓</td><td colspan="10">-</td></tr>
<tr><td colspan="2" rowspan="2">Way Limit</td><td>0</td><td colspan="6">Line Limit</td><td colspan="14">Index</td></tr>
<tr><td>1</td><td colspan="7">Line Limit</td><td colspan="15">Index</td></tr>
<tr><td colspan="32">↑ Small Data Process Address Space Tag (DPAST) ↑</td></tr>
<tr><td>0</td><td>0</td><td colspan="30">Invalid Translation</td></tr>
</table>

Table 12: Network System Reference (NSR) Field Descriptions

Bit Range	Field Name	Field Description
9:0 (LDO)	RESERVED	
28:10 (LDO) 28:0 (SDO)	Line Limit/ Index	This field contains the Line Limit and Index that the translation will insert into the DSR constructed by the translation mechanism. These fields have no corollary in the translated DPR and employ a simple substitution mechanism. The DPR index selects a DPAST→DSAST entry selects these fields independent of the LSO/SDO type of the translated DPR. Effective Offset size is the only limiting factor on DPR to DSR translation.
29	DSAST Size	This field determines, independent of the associated flag value in the DPAST being translated, the size of the DSAST and Offset in the translated DSR. Consistent sized translations are a simple replacement of the DSR's DPAST with the DSAST in the indexed DPAST→DSAST entry. An SDO to LDO translation is also trivial (the DPR has a smaller Offset then the DSR). If an LDO DPR is translated into an SDO DSR and if any of the upper 10 bits of the LDO DPR's Offset field are set, then an LDO cannot be constructed (translated) because there are not enough bits in the SDO's offset. A translation that experiences un-constructible SDO triggers a Programming Exception.
31:30	Way Limit	This field contains the Way Limit to be assigned to the DSAST. If this field's value is zero, then the associated DPAST→DSAST Table Entry is invalid and any DPAST that references it will generate a Protection Exception.

5.7.4 GAST-BAST-DSAST-DPAST Alignment

Storage is accessed via the selection of a system global set of [somewhat] persistent, Global Address Space Tags (GASTs) and its mapping to an ephemeral Data Process Address Spaces Tag (DPAST). This mapping requires that the system determine the requested GAST's BAST; if that BAST is not currently accessible it maps a DSAST to it. If it is already (currently) mapped to a DSAST, it uses its existing DSAST mapping. Once a GAST's transitive DSAST index is identified, the DPAST index allocated to the GAST is used as the index into the process' DPAST→DSAST in the CAS and the DPAST is pasted into the translation table, affecting a transitive GAST-to-BAST-to-DSAST-to-DPAST mapping. The GAST's access permissions, amended as required by the BAST's permission overrides, decorates the DPAST entry within the process' CAS. When the DPAST is encountered within an an address, the DPAST entry is pulled from the process' CAS by its PN into the PN's DPAST-to-DSAST Translation Table (`DP->DS_T`) or DPAST-to-DSAST Translation Cache (`DP->DS_C`), depending on the associated DPAST's index value.

A system's GAST vector contains the names of all the data objects available to a system and for each data object its assigned BAST and allowed access permissions available for that BAST via that GAST. The indicated BAST is an index into the system's BAST vector, which contains references to each data object's backing store and associated meta-data that indicated the object's current size, potential for growth, access permissions overrides, and more (see the BAST Structure in TBD for a complete enumeration of a BAST's metadata). A BAST represents potentially accessible data, but its associated data is only actually acces-sible if the BAST has been mapped to a DSAST.

An LLS maintains a DSAST vector that contains the inverse mapping from each DSAST back to its BAST and a forward mapping to each DPAST that can reference the DSAST. When the DPAST reference list becomes null, the associated BAST is physically inaccessible (even though it logically has a access vector via its current DSAST mapping, the fact that no process still has the DSAST in its DPAST→DSAST, makes it functionally inaccessible). The BAST's LLS can retire the DSAST at its leisure, though a modicum of urgency will improve the associated system's data integrity. A write through of all the dirty cache lines associated with a DPAST, and a discard of associated clean cache lines, are the preconditions associated with the termination of a DPAST-to-BAST mapping (the retiring of a DPAST).

If two processes shared data, then they have access to a common DSAST. There is no translation from a DPAST to anything other than a DSAST. There is no path from a DSAST to anything other than its mapped BAST. GASTs reference BASTs. BASTs do not logically reference GASTs (there is no interface that exposes BAST indices to users and hence, even though for management purposes the system maintains an inverse BAST-to-GAST mapping, this information is completely inaccessible to users (without access to BAST indices there would be no mechanism for a process to reference the inverse mapping even if it were otherwise accessible to a process)). there is no user interface that takes as an input or returns a BAST or DSAST. There is no user accessible storage structure containing a BAST or DSAST. There is no logical or physical mechanism via which a user process can learn a BAST or DSAST that is associated with its execution. Even if some form of divine intervention were to revealed the index of a BAST or DSAST to a process, the process would have no way of leveraging that information.

It is expected that only a small portion of a system's available data objects will be required at any given point in time. A system can have many more GASTs than BASTs because of the ability to have multiple independent GASTs that reference the same BAST. A many to one GAST-to-BAST mapping supports user identification and access permission differentiation. BASTs though an implementation artifact, are a useful abstraction for a system's data object set. BASTs (and their associated BASs) can demonstrate/employ process transcendent persistence. There is no DAST persistence (DPAST persistence it tied to

the life of a DPAST as expressed by its continued presence in any process' DPAST→DSAST.

A process can request the creation/generation of a new ephemeral data object, in which case a new BAST is created and mapped onto a system DSAST and on to a process' DPAST. Such an object is not necessarily tied to a GAST as part of its initial creation. When its [last] associated process terminates, if it has not been explicitly discarded or explicitly associated with a GAST-to-BAST mapping, it is automatically discarded. All the cache lines allocated to a discarded DSAST are scrubbed from the SAFE CH and any LLS local content is freed. A BAST without an associated GAST has no reference tag that survives its [last] associated process (there is no way for a user to identify it). A SAFE system's internal management subsystems can usefully employ ephemeral BASTs for a process' stack, heap, and/or any other transient information storage.

At DSAST creation, an existing BAST is treated as a copy-on-write object. This preserves the BAST's initial data state until a process successfully closes and successfully updates all the changes made to a data object. This supports the ability to fall-back to a previous version in the event of an error and/or process failure. Multiple DSASTs can be treated as an update group with all group members or none of a group's members being concurrently updated. On process termination or DPAST closure, a DSAS state must either (1) discard all the modifications associated with a BAS; (2) integrate the unchanged portions of the original BAS into the modified version of the BAS, thereby creating a new data object; the original BAS is discarded and the new version assumes the BAST of the old object; (3) the updated/integrated BAS is constructed and the original BAS is preserved; the new version of the BAS is assigned a new BAST and GAST; or (4) the updated/integrated BAS is constructed and the original BAS is preserved; the new version of the BAS is assigned the original BAS's BAST and the original BAS is assigned a new BAST and GAST.

5.7.5 DAS Programming Model

A process code base that is executing within a DPAST-based memory model will function completely unaware of SAFE's existence. A SAFE pointer and a legacy pointer are both 64-bit numbers that reference data. SAFE's AST model is wholly congruent with existing memory allocation mechanisms. The legacy memory model relies on the assignment/ allocation of addresses that are logically base address / limit allocations, but that functionally are just memory mappings. exactly of the sort SAFE's DAST model supports. Both legacy and SAFE process has unrestricted access to its addresses (pointers). A legacy process generates a Segmentation Violations if it touches an unallocated page (one not supported by the process' page table), or a Stack Overflows if its stack attempts to overwrite its heap. SAFE's Protection Exceptions, which can be made to look like legacy exceptions, are triggered by a access to an unmapped PAST or and overflow of a valid PAST[1]. The SAFE model is no different from the legacy addressing model in the sense that there are valid and invalid address ranges and touching an invalid range kills the touching process. SAFE has the advantage that read-before-write is guaranteed to be at worst benign (partially filled cache lines are nulled and an exception is generated if a process touches an invalid cache line).

A process' DPAST-to-DSAST Translation mechanism does not exist from the process' perspective. Processes are given pointers and those pointers have limits (from a process' perspective and system management model perspective, limits only become relevant when they are violated). The association between pointers and data objects is easily seen by an interrogation of the associated pointer, but from a security perspective, there is nothing meaningful that a process can extract from such information (knowing that a data object is mapped to a specific address range does nothing to impact the options available for interacting with the data object). At some point, a more SAFE aware PN could usefully fault if a PN's manipulation of an address results in a reference to an address in a different DPAS (such checking/monitoring would be very useful addition to system functionality, but it would require changes to a processor's Instruction Set Architecture (ISA) that differentiates address addition/subtraction from those not unassociated with address generation. Such a change would be useful, but it is not neces

DPAST-to-DSAST translation (`DP->DS`) is akin to TLB based virtual-to-physical address translation. Unlike a TLB, `DP->DS` supports an accelerated table based mode. Fast `DP->DS` employs a directly indexed table (`DP->DS_T`) that supports access to a specific range of DPAST translations. Slower `DP->DS` employs a cache (`DP->DS_C`) to support access to the full range of non table-based DPASTs. A SAFE implementation need not support a `DP->DS_T` (`DP->DS_C` implementation is mandatory, short of building a 64_512+192 entry `DP->DS_T`). If a `DP->DS_T` is implemented, the range of DASTs covered by the table is an implementation decision. It is strongly suggested that at least SDO `DPAST[0]` and `DPAST[1]` be table based. A reasonable approach, and the suggested mechanism for standard implementations, is to support a 16-entry SDO table that supports direct lookup of Small DPASTs 0:15, and a 4-entry LDO table that supports direct lookup of Large DASTs 64:67. The size of the required cache is implementation dependent, with the option of a single entry cache (for many processes, a table based translation of `DAST[0]` and `DAST[1]` and a two-entry cache would have reasonable performance, though for many other processes such a structure would be found lacking - the cost of a table is small and its power savings are significant, within the modest scope of TLB-ish power consumption). A 4-way, set associative cache with at least 64 entries is the suggested minimal `DP->DS_C`s implementation, with 256 to 1024 entries a reasonable option for forward looking SAFE aware designs.

Both table and cache are demand loaded. On start-up a process is not stalled by its DP->DS state (except for `DPAST[0]`, standardized as the process stack, which will stall the associated process' execution until it

1. Protection exceptions are experienced by a process as a delayed response that forces the process from its PN. Once passivated, the associated PMU discards the process with appropriate notifications and logging.

has been automatically loaded as part of PN's process context recovery). Other DP->DS state is recovered as an artifact of process execution. Using modern (circa 2020) silicon feature sizes, an L1 cache is expected to require a bit less than a single nanosecond to access. A table-based translation will have no impact on this access latency (the bits required for an L1 tag comparison will be available before the tag will be - this is true even for relatively large tables). A cached translation may have a small impact on L1 latency (no worse than a TLB's[1]). A miss in either table or cache can have a considerable impact on access latency, requiring a CAS read that may or may not already be cached in the node's L1 cache (impact of a DP->DS miss is directly tied to the cache depth to which the associated read must descend to locate the CAS data that contains the requisite translation, though here again, it will be no worse than current TLB miss behavior, and generally better). DP->DS miss behavior is analogous to TLB miss behavior, but with the advantage that a single DP->DS entry can translate up to a pebibyte of data, not the 4kB that a TLB entry translates. Dozens of DP->DS entries are generally more effective than thousands of TLB entries (a TLB can concurrently handle substantially less than a tenth of a percent of a modest size system's memory, with exclusive usage of large (2mB) page indices being required to get the percentage to 10%). Fewer than a dozen DP->DS entries could easily translate an entire system's memory footprint. The complexity of TLB misses that require multi-level, hardware-based, page-table walks can be abandoned in favor of a simple direct `DP->DS_C` access or single-level `DP->DS_T` lookup.

DSAST line and way limits allow for the creation of a significant number of unique DSASTs, while concurrently, selectively restricting the resource consumption of the constructed DSASTs. This is SAFE's top-down resource constraint model. There is a companion bottom-up resource constraint model that is managed by the LLS. It is easy for the LLS to track the number of lines in a system's L5 that are associated with a given DSAST. Limits on the number of such lines can easily be imposed by an LLS such that when the limit is triggered, an L5 DSAST cache line culling can be performed. The LLS does not have access to the LRU status of the L5 and the L5 is not indexable by DSAST. Given that L5 cache lines can be retired in an order incongruous with their initial request order, maintaining a dynamic FIFO order can be expensive. The proposed L5 cache model employs a Not-recently Used (NRU) model that walks the cache to locate replacement targets. It is relatively straight forward to integrate a DSAST scrubber into this NRU model. A relatively small vector of replacement targets could be freed if they are encountered during the NRU's cache walk. A counter associated with each target DSAST entry in the L5 cache scrubber indicates how many remaining associated L5 cache lines must be evicted. When a counter goes to 0, a new DSAST replaces it in the L5 Scrubbing Vector. This model allows brief over-superscriptions (the size of the L5 and its management mechanism makes brief subscription an non-issue).

If an LLS's per DSAST L5 Maximum Line Count (L5-MLC) is violated, the LLS will request the removal of 2 to 8 of the DSAST's cache lines (depending on the size of the associated MLC). An L5 receiving such a request will prioritize the eviction of lines that satisfy the system's NRU mechanism. Unfortunately, the removal of an L5 cache line may necessitate its removal from all higher level caches as well (or the conversion of the cache line to Null status (discussed later). An evicted L5 with an L4 presence necessitates the forwarding of a discard or discard with write-through request to its associated L4(s). If the L4 being evicted has an L3 presence, then the L4 passes the discard request up the CH until all higher level cache content has been flushed or discarded.

The combination of top-down and bottom-up DSAST resource allocation management will provide a level of resource management not significantly different in effect from that supplied by legacy OS software, but without the need to context switch to the system's software-based management model. The SAFE model is orders of magnitude faster and

1. A `DP->DS_C` of modest size (64 entries) is more effective than relatively large TLB (1500+ entries) and in most cases a `DP->DS_C` will have superior performance (both power consumption and latency) than the TLB it replaces.

lower power, while also creating a significantly more secure system [Note: this efficiency claim is relative to a management mechanism that is used infrequently and hence it will have very little effect on a system's overall performance].

An implementation can employ a compressed DSAST when the full range of DSASTs is not deemed necessary (viewed as supportive of a range that is significantly larger than is required by the target system). In these cases, the underlying hardware simply imposes implied 0 bits in the low order bits of an DSAST's Line-Limit and/or it imposes implied 0 bits in the upper bits of the Index field, thereby reducing the size of the associated hardware infrastructure accordingly. For example, if 2 bits of Line Limit and 6 bits of Index are dropped, the required tag's size is shortening by 8 bits.

SAFE's Input/Output Model (SAFE-IOM) makes each PN (CPU or accelerator (dynamic and/or fixed functional unit) and SC (LLS, PMU, AE, SNE, or SAE) a largely-independent processing element that communicates with its associated PNs and/or SCs via SM and MNs. As a general statement, it makes sense for an I/O device to size shared memory allocations to match the associated communication bandwidth requirements. Balancing input/output buffer requirements, with the option of dynamic and flexible buffer consumption, provides a form of self governing communication flow.

5.8 Network Address Space (NAS)

A NAS is a linear stream of bytes. It has one sender and one receiver. Multicast (the ability to simultaneously distribute a single data stream to multiple receivers) is not supported by SAFE's NAS infrastructure[1]. A multi-writer and/or multi-writer-multicast capability is also not supported[2]. Any network traffic that does not transcend a single SNE does not enter the NAS domain, as such, and is not overseen by SAFE' infrastructure[3]. A SAFE system can be configured for trusted inter-SNE network flows that employ DAS resources in place of NAS resources (NAS resources have a somewhat higher management cost that can be avoided if correctness is relaxed between System Components (SCs). Conversely, the NAS infrastructure is as applicable to inter-SNE traffic as it is to SNE-PN traffic and there is no need for a relaxation of trust.

A process uses a Network Process Reference (NPR) to access a Network Process Address Space (NPAS), which is mapped to a Network System Address Space (NSAS). An NPR is used to generate a Network System Reference (NSR) via the translation of Network Process Address Space Tags (NPASTs) to Network System Address Space Tags (NSASTs) . The underlying sockets that these ASTs represent support unicast data flows between an SNE and a process or between a process and an SNE. A network flow is associated with a process (CAS), not the PN on which a process is executing because: (1) the process may not be executing when it receives traffic, (2) the process may not be executing when traffic it has previously scheduled for transmission is actually sent, and (3) the process may not resume on the PN on which it previously executed. Network flows between System Components (SCs), of which an SNE is a member, uses DASs and/or SM with MN-based notifications.

SAFE's internal NAS data flow mechanism is physically a stream service. Datagrams can be constructed as a logical layer above the stream interface via a modicum of discipline (adherence to a data and messaging protocol) on the part of its users, or, preferably, via the strict use of a datagram library. A socket can be used as a stream and/or a datagram (or any other system specific data transport model that is layered on the hardware supported stream model), though adherence to a socket's indicated usage model is not guaranteed by the default SAFE hardware infrastructure (there is no way to guarantee that a process will us a specific function to process a data flow). Logically, this is not significantly different from any range of errors a process can commit. The use of a network-specific service interface via a SAFE Binary Adaptation Layer Library (BALL)[4] that is tied to a legacy DLL/library can offer a modicum of correctness, but not a guarantee (the underlying socket interface

1. A SAFE system can support multicast via a multicast proxy that acts as a single socket receiver and then uses a DAS and/or Shared Memory, both of which require Messaged Notifications (MNs), to distribute network traffic to all receiving sockets. The proxy need not logically be a member of the multicast group, that is, the target process can purely function as a proxy for the multicast group, or for a number of related or unrelated multicast groups. Depending on the nature of the multiocast group, a proxy may choose to employ multiple NASs to effect its operation (a single SNE can abstract multicast on one or more of its physical networks, but a multicast that targets two or more independent SNEs has a range of potential proxy models).
2. These are much less common socket configurations, but as with a multicast connection, a process can act as proxy for an arbitrarily complicated socket network.
3. A NAS that is configured for packet forwarding via one or more of its captive physical network connections does not consume any shared system resources and is therefore transparent to the associated SAFE system. This includes the actions of a compromised SNE whose operation is localized to its captive resources. A network monitoring applications is a highly suggested service module for a system, but the operation requires the imposition of a network monitoring capability that can communicate independently with a system, for example, via USB on an independent SNE, that has a packet and bandwidth counting capability between an SNE and its associated network switch(s). Additionally and/or alternatively, a power monitoring system system can also detect any significant delta between an SNE's reported activities and its real activities.

is still exposed as an element of the associated processes address space and it can be erroneously accessed by an addressing error).

Each NSAST index is associated with a physical SNE and each physical SNE has an associated 1024^2 aligned, linear vector of NSASTs (an SNE controlled number of which are valid at any given time). An SNE can only communicate with NSASTs that fall within its range (an SNE that generates a NSR with anything other than its assigned SNE Index will generate a Protection Exception, which stalls the SNE and typically results in the SNE being reset (this is a serious error that generally indicates a significant SNE failure and a reset will force an SNE self check). A reset SNE will lose all of its existing connections. A failed SNE cannot be allowed to initiate additional system traffic. An SNE's NSAST usage is not affected by the execution of its associated user processes (an NSAST range error cannot be caused by a process error or process shenanigans[1]). A NPR-to-NSR translation is associated with the each valid NPAST. An NPR-to-NSR translation involves two independent and parallel infrastructures (as described in "NPR-to-NSR Translation" on page 57).

Bytes read from and written to NASs are ordered based on the size of the associated data operation. Most CPU architectures either have a fixed or a configurable endian-ness, but seldom a write specific endian-ness. For efficiency reasons, a multi-data write that has a network-esque (size, count) tuple is valuable. To address this general deficiency, SAFE's NSAST configuration includes a trio of process mutable bits. NAS reads and writes are inherently unaligned, with configurable data transport characteristics, which can be modified via the SSR interface. A vector of per-socket SSRs supports the management of atomic data size and endian-ness (big-endian/little-endian, 1-byte/2-byte/4-byte/transaction-sized). A reading process can optionally write to its input buffer, and reading and writing share a single NPAS data order configuration (it writes back into an NPAS in the same order as it came out, unless the configuration is changed between read and write).

While somewhat sophisticated, the requisite hardware associated with NAS state and behavior management represents significant savings relative to a legacy network stack's behavior. This is particularly true regarding a significantly heightened security the SAFE offers, particularly when the significant performance advantages of its user level interface are factored into the model. All the capabilities of SAFE's networking interface will not be exercised by legacy binaries, but SAFE will provide important capabilities to next generation code bases.

An L4 can be partitioned between its DAS and its NAS cache blocks. The flexibility of a partition boundary is an implementation decision. L4 caches constructed from low-endurance storage technology will require the addition of a high-endurance buffers (DRAM) that can hold short persistence data from high-traffic NASs (high volume data transactions are typically of the forwarding variety and require an active (running, not ready or blocked) process to prevent buffer overflow). Flowing such data through highly resilient and low latency L4 buffers will significantly reduce wear on lower endurance storage/memory technology because network buffers are generally shorter lived that memory/storage buffers.

5.8.1 Partial Cache Line Writes

To improve the efficiency with which NAS cache lines are handled, SAFE employs its *Partial Cache Line Update* model (also used in a more restrictive mode by the DAS infrastructure) to reduce data movement overhead by only sending the modified portion of a cache line from writer to reader. For NAS writes, partial updates are used exclusively. *Write combining*[2] and *delayed forwarding*[3] are also employed to minimize cache transaction count.

4. A BALL is an operating system specific application wrapper that creates comparability between an existing application binary and SAFE's containerized hardware interface.

1. Process Shenanigans is a technical term that refers to a process purposely generating incorrect addresses and/or data in an attempt to perturb system state.

NAST cache line updates can be routed directly to peer L3 caches using a special case of Null Caching[1]. The partial cache line write model pushes cache content directly to its associated L4 cache, and potentially without being stored in the L4, up to an associated target's L3, L2, and L1 cache lines (data is pushed into its owning locus within the CH). Pulling the cache line from its target to append to the line and then send it back to the target is inefficient relative to simply sending the update to its targeted cache line(s).

NAST storage does not exist below the L4 level. A PMU performs double duty as connection manager and L4 control agent for its NASs. If an L3/L4 cache starts to run out of storage space for NASs, a PMU and/or an SNE can pull data from its NAS buffers and store them into one its own PDAS objects, retrieving the buffers at some later point when the explicitly stored buffer state is needed.

A NAS reading process' can request data from an L4 cache and the L4 can recognize the request as being Null Cached and forward the request to the appropriate L3, which will then respond directly to the requesting L3. The data associated with Null Cache transactions does not need to transit the associated L4(s). A NAS cache line that is evicted from an L3 may require allocation in an L4 if it is located in a Null Cache.

5.8.2 NAS Writes

When a process is writing to a NAS (socket). The written data is appended to the NAS using its CAS-based Target Socket Table entry. A writer can only supply a NPAST (no associated Offset because the address in the translated NPR write is supplied by the auto-incrementing Tail Pointer in the associated process' (CAS's) socket translation table and cached in the PN. That is, every byte written to a socket is appended to the end of the associated stream at the location referenced by the Tail Pointer and the Tail Pointer is incremented following each byte write. This is a system managed mechanism over which the associated process has no control (aside from deciding not to use it). The only "control knob" provided a Target Socket is a per Target Socket SAFE State Register (SSR) that indicates the number of additional bytes the associated socket can absorb before the associated buffers are full (a socket overflow typically kills the writer and sends a notification to the reader).

The Offset is 32-bits in size and it simply wraps within its 32-bit range, with the wrap representing a step into the next 2gB aligned region of the associated NAS's absolute offset. The Offset is incremented per byte written to the BAS. NAST writes allocate cache as necessary, unless a write violates the NAS's configured maximum size. Writes to the L1 cache line associated with the NAS's current offset proceed as they would any other cache write, and like any other cache write the update is relatively silent, from a system perspective. A socket writer is logically the inherent owner of the cache line being written, however, that the Partial Cache Line Write mechanism pushes ownership towards the reader, making the question of cache line ownership highly transitive. The optionally enabled reader writing requires that cache line merging logic be implemented in the L1s to update only the newly writer written bytes (pushed partial cache line write data) to the cache line (this is a very simple mechanism).

---

2. The initial level of write combining is done within a PN's infrastructure and involves the concatenation of a series of writes to sequential memory locations into larger operations (for example, depending on alignment, 8 single bytes writes may be combinable into a single 8-byre write).
3. Delayed forwarding inserts a small delay in the forwarding of a network block in the hope that subsequent data will be forthcoming. In general, delayed forwarding will attempt to collect entire L2 cache lines before sending a block. The presence of an alternative network flows in the same physical channel, will trigger the forwarding of delayed buffer.
1. Null Caching is discussed in detail later. For now, at a high level, a Null Cache is metadata state maintained at a cache without any associated data. SAFE's inclusive caching model would otherwise require that higher-level cache state be evicted if it is evicted from a lower level cache. For normal data, only clean cache lines can be placed in a Null Cache. NAST cache lines allow dirty upper level cache lines (made easier to handle by the single writer restriction on NASTs).

5.8.3 NAS Reads

The reading side of a socket can be configured for serial access or buffered access. Serial access is the compliment to the NAS write mechanism: reads are performed without an active offset field and each read returns the next byte(s), relative to the associated sockets Head Pointer (as indicated by the associated sockets, Source Socket Table entry in the associated process' CAS).

In buffered mode, received data is extracted from an indexable buffer - the logical 0-base location of the buffer is mutable and a process can read any active buffer location relative to its Base Address[1]. A process controls the location of an input socket's Base Offset via an Advance Base [Offset] (AB) flag within an NPR. Setting the flag repositions the associated input buffer's Base Address to the location one byte beyond the last byte read. Reading with a cleared flag does not affect the value of the NPAS's Base Offset.

Input is read using a 32-bit current Base Address relative Offset field that is used to generate a 40-bit Offset into the associated NAS's 1tB ring buffer[2]. The 32-bit read supplied offset is added to the 40-bit Source Socket Base Address field to generate a 40-bit NSR Offset via the Source Process Socket to Source System Socket translation infrastructure. An NPR that touches an invalid cache line (the referenced data has yet to be written by the associated NAS's source) results in the generation of a Protection exception (standard within the last cache line reading of default 0 values for uninitialized data applied).

Readers can move through the NPAST buffer at a self defined rate, re-reading the buffer at their discretion and moving the buffer's base address forward at their discretion. As its NPAS relative Base Offset advances across L1 cache line boundaries, the associated PN's SAFE management interface discards the no longer addressable L1 cache line (a base address cannot move backward). The L1 discard notification sent to the L2 will trigger an associated L2 discard if the discarded L1 was the last partition of the associated L2. The L2 discard moves to the L3, which can move to the L4 (the last cache in which network traffic is cached). This is all standard cache behavior, nothing special for NASs, relative to DASs, except for the terminal behavior at the L4. Based on the aggregate state of all its associated readers, an L4 cache line that contains no longer accessible data, at the system level, is discarded; unlike DAS cache lines that flows through to the associated L5 and eventually to the LLS.

[Note: while this model is fully compatible with the existing socket model that requires a copy of network stream contents from network buffers to "user space" buffers before the buffer's contents can be inspected, the SAFE mechanism also supports in-place data access by targets, including the ability to configure a socket's input buffer for writing (a process can overwrite its input network stream's contents within the stream itself[3]).]

When a process is reading a NAS (socket), the data is presented in standard L1 cache lines. These cache lines can be configured to support writing (the associated process can update data already received via a network connection in situ). This model obviates the need to copy data into local buffers before acting on them. This model also allows data to be interrogated within the input buffer and its disposition to be deter-

1. A process can also read invalid locations, up to the point that it generates a page fault that is known to resolve to a cache line not yet written by the other side of the socket.
2. An NSAST's offset range is a monotonic increasing index into a logically defined 1tB buffer that can never physically be larger that 4gB because offsets become invalid as the NPAST's window slides across the NSAST offset range, and that can be configured to be smaller than its maximum possible range (it can be configured to be as small as 256 bytes).
3. Only the active (previously received) portion of the stream can me updated by its target. Any attempt to write to a input buffer location that has not yet been validated by the receipt of data from the associated socket's writing process will result in the generation of an Protection exception.

mined before it is positioned in process specific memory. It also significantly reduces the overhead and associated memory footprint required for forwarding buffers that require minor modifications to their content before they are sent.

This mechanism gives a socket writer primacy over its associated readers by unconditionally overwriting reader cache line updates with new writer generated data (a writer knows where the logical end of a buffered socket's data stream is within its associated NAS, but nothing prevents it from violating this boundary within the confines of a partial terminal cache line, all the way to an L2 or L3 cache line boundary). However, any data placed in this gray area is summarily replaced by socket writer supplied data (a socket reader that writes past the end of a buffer looses any data written to future writer generated updates).[1]

Sockets operating in write-ahead mode (when a socket's data consumer reads, the socket's writer has already placed at least some data in the buffer) or in read-ahead mode (the socket's data consumer reads the socket before data is available from its writer). Write-ahead keeps a process in forward-progress mode and makes blocking an optional decision (read, if insufficient data is available, read again, and continue until sufficient data is available; or, if no data is available, block). Read-ahead mode relies on the system's MN mechanism.

A synchronous socket reader (NAS target) blocks on the MN queue associated with the NPAST if it wants data, but the desired data is not yet avaialble. After posting data to a socket, a socket writer updates the NAS's Tail Count by the number of bytes added to the associated NAS's buffer). A process does not block on an NPAS, per se. It blocks on the MN queue that is used to communicate NPAS state changes. Hence, a process with dozens of open sockets can use a single queue that receives updates for all its connections and manages process state accordingly. [Note: while this model is fully compatible with the existing socket model that requires a copy of network stream contents from network buffers to "user space" buffers before the buffer's contents can be inspected, the SAFE mechanism also supports in-place data access by targets, including the ability to configure a socket's input buffer for writing (a process can overwrite its input network stream's contents within the stream itself[2]).

5.8.4 Socket Messaged Notification Model

NAS data flows rely on a socket writer keeping its peer socket readers aware of network data movement via MNs (an SNE sends a message to the process to which it is writing and a process sends a message to the SNE to which it is writing). Once a peer has been notified via an MN, additional MNs are blocked until the socket's reader enables notifications via a write to . When the targeted PMU has handled the MN and a reader has responded to the presence of new data, the PMU sends an MN back to the process that unblocks NAS associated MNs. This MN throttling decreases the incidents of what would functionally be unnecessary/spurious notifications, and helps to keep PNQ depths manageable.

In concert with MN based connection peer Notification, a NAS writer also performs a series of virtual NAS connection peer register updates. NAS targets have access to a NAS source owned shared memory buffer, via the SAFE State Register (SSR) interface (this is not the standard shared pair-wise buffer sharing the SAFE allows, but rather a hardware

1. Preventing a process from writing into its input buffer past the logical end of that buffer is a superior solution, but the overhead of checking is not worth the value of checking. Such errors fall within a set of wide ranging software errors from which SAFE cannot provide primary protection, but which safe prevents from cascading into secondary security issues. In this case, a the associated error will trigger a Protection exception if the associated library is allowed to execute for any extended period.

2. Only the active (previously received) portion of the stream can me updated by its target. Any attempt to write to a input buffer location that has not yet been validated by the receipt of data from the associated socket's writing process will result in the generation of an Protection exception.

mediated and controlled mechanism for reading another process' CAS state). In this case, a writer's NAS Tail Pointer register, written by a writing process to its NPAS specific SSR, is made readable (not writable) via a reader's NPAS specific SSR. This is "behind the scenes" behavior enabled by SAFE state that is not available at the process level, except via carefully controlled data exposed via SSRs.

The pair of SAFE State Registers (SSRs) per NPAS writer that have varied interpretations based on the configuration of the associated network connection. The writer's registers are:

- Get Head Pointer- The 40-bit absolute system Offset of the associated socket's Head Pointer is returned. The returned value cannot usefully be used to access the associated NAS (write addresses are automatically generated, though when subtracted from the associated Tail Pointer it indicates how much data is the NAS).
- Get Tail Pointer - The 40-bit system Offset of the associated SQ's Tail Pointer is returned. The returned value cannot usefully be used to access the associated NAS (write addresses are automatically generated, though it does indicate how much room is left in the AS.

The reader's registers are:

- Get Tail Pointer- The 40-bit absolute Tail Pointer of the associated NAS writer is returned.
- Get Buffer Size - This is the number of bytes in the associated reader's queue. This value is dynamically generated by subtracting the NPAS reader's 40-bit Head Pointer from the associated NAS writer's Tail Pointer.

The blocking of MNs associated with a NAS writing process does not stop the updating of the NAS's Tail Pointer, communicated by SSR writes. Hence a process that has been awakened, or otherwise triggered by the delivery of the initial MN associated with a NAS update, will see the current location of the NAS's tail and be able to process all the associated data, without additional Notification.

In addition to default MN throttling behavior, a writer's NAS specific MN triggering can be disabled or paced (it can be configured to deliver MNs at no more than a specific configured rate - once disabled, it will not be enabled until a specific quantum has expired and it would otherwise have been triggered). This behavior can spread out the rate at which a process handles its network flow, but with the caveat that a sockets data rate is functionally disconnected from its MN flow (the process' PNQ high-water will only help if is set to a low value).

An NPAS that violates its configured maximum buffer size will send an MN to its PMU and, if the writer, the associated SNE. A process will receive a standard socket protocol "socket-closed" signal if one of its network streams is closed for a buffer violation and the process will take its programmed remedial actions, if any (as is the case with standard (legacy) process management, an unhandled signal will terminate its associated process). A close of the connection's (socket's) last reader will close the connection's writer (with symmetric "SIGPIPE"/"EPIPE" signaling).

5.8.5 Datagram Model

As previously indicated a NAS is inherently a data stream. The imposition of a datagram model is a matter of discipline and software managed protocol. SAFE datagrams are tagged streams. A socket that carries datagrams can also carry stream data, but it must be tagged as if it is datagram traffic. SAFE's network transport is inherently reliable and hence there is relatively little difference between streams and datagrams. The MN model that manages stream sockets is equally applicable to datagrams. A SAFE datagram is composed of op to 16_384 segments with up to 65_536 bytes per segment (maximum of 256mB per datagram).

A datagram source supplies the 16-byte IPv6 target address, a 2-byte port index, a 2-byte segment index (the most significant bit of which indicates an empty data payload, the second most significant bit of which indicates the terminal segment of a block, the next 2 bits of which indicate a segment index, and the last 12 bits of which indicate the associated block's segment index), a 2-byte data payload size, and 2 bytes of pad. Datagrams are aligned to 8-byte network stream boundar-

ies. A datagram target reads a data flow that is tagged as indicated by a source. A stream can be overlaid on a datagram flow. Datagram payload indices sequence from 0 through 2 and back to 0. A stream within a datagram socket used payload index 3.

Table 13: Datagram Format

Word	7	6	5	4	3	2	1	0
0	IPv6[63: 0]							
1	IPv6[127:64]							
2	Pad		Payload Size		Segmentation		Port Index	
3	Payload Word 0							
4	Payload Word 1							
5	Payload Word 2							
N+2	Payload Word N -1							

Table 14: Datagram: Segmentation Format

15	14	13	12	11	10	..	1	0
NP	T	Datagram Index		Segment Index				

NP = Null Payload **T** = Terminal

5.8.6 Network System Reference

A Network System Reference (NSR) is composed of a Source/Target flag, a 28-bit Network System Address Space Tag (NSAST) (8 bits of Secure Network Engine (SNE) Index, 20 bits of Socket Index), and a 40-bit relative Socket Offset.

Each SAFE process/thread can concurrently interact with up to 16 SNEs, with up to 4096 socket connections per SNE (a total of 65_536 concurrent connections). A process that requires a higher network interface count and/or socket count can be constructed from a vector of threads. A process that wishes to have more than 4096 concurrent connections to a single SNE can configure the translation interface to employ a shared SNE top effect an extension of the Socket Index field. All 65_536 of a process' potential Socket Indices can referrence the same SNE.

The translation processes' Process SNE Index (PSNE) to physical/System SNE Index (SSNE) is controlled by a PSNE-to-SSNE translation table (`PN->SN`). As previously indicated, a `PN->SN` supports the blurring of the SNE Index / Socket Index distinction by allowing multiple PSNEs to reference the same SSNE, with an extended Socket Index. The expansion field for NPRs is the SNE Index, allowing a process to expand it socket count via the use of additional SNEs.

An NP's SAFE Adaptation Layer includes a 16-entry `PN->SN`. and a vector of per Socket Index metadata that includes: the socket's the data model (base data unit width and endian-ness) and current Base Address. An NPAS Reference (NPR) represents a dynamically sized, sliding window into a NSAS. A NPR's Offset is relative to the position of the NPR's queue's head pointer. A NPR can concurrently access, at most, a contiguous 4gB region of its associated source NSAS's 1pB NAS. The 4gB window is relative to a monolithic increasing NPAS Base Address whose location is controlled by the Advance Base (AB) flag in an NPR.

Table 15: Network System Reference (NSR) Format

71	70	69	68	67	66	. . .	61	60	59	58	57	. . .	42	41	40	39	38	37	. . .	2	1	0
0	0	0	ST	Network System Address Space Tag (NSAST)												Offset						
				SNE Index					Socket Index													

SNE = Secure Network Engine **ST** = Source (0) / Target (1)

Table 16: Network System Reference (NSR) Field Descriptions

Bit Range	Field Name	Field Description
39:0	Offset	This field contains the Offset within the indicated NSAS's ring of the associated reference. A socket's buffer wraps within a maximally sized 1tB ring (it can be configured to a much smaller size).
67:40	Network System Address Space Tag (NSAST)	This field contains the 28-bit NSAST of the associated NAS. This is a system-wide identifier that is managed by its associated PMU.One side of a network connection (socket) is attached to a process and the other end is attached to an SNE), an embedded networking subsystem. It is possible to connect intra-system SNEs with a socket, though the use of the MN and shared memory infrastructure is more efficient.
59:40	NSAST: Socket Index	This field contains the socket index with which the associated network traffic is associated. A socket can have an associated bandwidth limit that generates a MN if more that the indicated number of bytes transit a socket. Bandwidth limits are unidirectional (there are two limits per socket, one per direction).
67:60	NSAST: Secure Network Engine (SNE) Index	This field contains the Secure Network Engine (SNE) Index of the associated socket's external connection. Sockets are allocated in 1024^2 blocks to SNE Indices, with the option of allocating multiple blocks to individual physical SNEs (thereby functionally extending the Socket Index into the SNE Index). NAS routing from an SNE checks NSRs to ensure that an SNE has properly identified itself. NAS traffic heading towards an SNE uses this field to target an SNE.
68	Source / Target (ST)	A target referent sends data (writes) to the socket. A source reference retrieves data (reads) from the data object. This flag, when set, indicates that the accessing process is writing to the associated socket. When clear, this flag indicates that the associated process is reading from the associated socket.
71:69	Must be 0	This field's value must be 0 for the associated SR to be interpreted as an NSR.

5.8.7 Network Process Reference (NPR)

Network Process References (NPRs) are differentiated as referencing either a source (SNPR) or a target (TNPR) flow[1]. TNPRs are append only and have no associated offset field (the Offset used in the construction of an NSR is the associated socket's current Tail Pointer. A TNPR is an NPAST (a 4-bit SNE Index and a 12-bit Socket Index). A SNPR is a NPAST, an Advance Base (AB) flag, and a 32-bit relative Socket Offset. NASTs are the purview of PMUs.

1. Source and target distinctions are relative to the action applied to the AST itself. A writer appends data to a data target and a reader pulls data from a data source.

Table 17: Source Network Process Reference (NPR) Format

57	56	. . .	51	50	49	48	47	46	45	44	43	42	41	. . .	34	33	32	31	30	. . .	1	0
1	1	1..1	1	1	0	AB	Network Process Address Space Tag (NPAST)											Offset / Reserved				
							SNE Index				Socket Index											

AB = Advance Base Offset (1) / preserve base (0) **SNE** = Secure Network Engine

Table 18: Source Network Process Reference (NPR) Field Descriptions

Bit Range	Field Name	Field Description
31:0	Offset / Reserved	When the associated socket is configured for Head Pointer operation, this field is reserved (the read address is implied by the socket translation's Head Pointer and cannot be specified). This field is ignored, though good form requires it to be zero (failure to maintain goos form does not generate an exception). When the associated socket is configured for Buffering, this field contains a relative forward (positive/ unsigned) offset from the socket's current Base Address (as indicated in the NPS→NSS). An NPAST Offset range is composed of a pair of 2gB blocks. The first block (most-significant bit clear) can logically precede or follow the other 2gB block (most-significant bit set). This field contains the relative Offset within the NSAST to which the NPAST is mapped, but relative to the 2gB block in which it resides. The associated process' NPAST→NSAST contains a base address. Bit[31] of the associated NAS's Base Address indicates the relative 2gB block order associated with this field (if the bit is set, the logical order is reversed).

Table 18: Source Network Process Reference (NPR) Field Descriptions

Bit Range	Field Name	Field Description
47:32	Network Process Address Space Tag (NPAST)	This field contains an NSAST target-specific 16-bit index for the associated access. This is the process-specific identifier that is translated to an NSAST by the executing PN's hardware accelerated NPAST-to-NPAST Translation Table and Cache, which in turn are dependent on the process' CAS-based NPAST→NSAST. In general (SNE to SNE connections are also supported), one side of a network connection (socket) is attached to a process and the other end is attached to a Secure Networking Engine (SNE). An SNE to SNE socket can more efficiently use DAS, MN, and SM infrastructures.
43:32	NPAST: Socket Index	This field contains up top 12 bits of NPAST mask for NSAST construction and the lower 12 bits of the offset into the CAS's 65_536 Target Socket Translation Table.
47:44	NPAST: Secure Network Engine (SNE) Index	This field contains the process-specific index of the socket's associated SNE. This field references an entry in the associated CAS's NPAST→NSAST, which in turn is used to extract a System SNE and a Socket Index mask for construction of the associated NSAST.
48	Advance Base Offset (AB)	This flag, when clear, indicates that a source access of the associated NPAST does not advance the associated socket's base address (none of the associated cache lines can be discarded). When set, this flag indicates that the a source access does advance the associated sockets base address and that it advances past a cache boundary the cache line(s) that are no longer within the sockets active window can be reclaimed (discarded). This flag only applies to sources (sockets that can be read). This flag is reserved when associated with a target.
57:49	Must be 0x1FE	This field must contain the value 510 (0x1FE) for the associated reference to be interpreted as a NPR.

Table 19: Target Network Process Reference (NPR) Format

57	56	. . .	44	43	42	41	40	. . .	17	16	15	14	13	12	11	10	9	. . .	2	1	0
1	1	1..1	1	1	0	1	1	1..1	1	1	Network Process Address Space Tag										
											SNE Index				Socket Index						

SNE = Secure Network Engine

Table 20: Target Network Process Reference (NPR) Field Descriptions

Bit Range	Field Name	Field Description
15:0	Network Process Address Space Tag (NPAST)	This field contains an NSAST target-specific 16-bit index for the associated access. This is the process-specific identifier that is translated to an NSAST by the executing PN's hardware accelerated NPAST-to-NPAST Translation Table and Cache, which in turn are dependent on the process' CAS-based NPAST→NSAST. In general (SNE to SNE connections are also supported), one side of a network connection (socket) is attached to a process and the other end is attached to a Secure Networking Engine (SNE). An SNE to SNE socket can more efficiently use DAS, MN, and SM infrastructures.
11:0	NPAST: Socket Index	This field contains up top 12 bits of NPAST mask for NSAST construction and the lower 12 bits of the offset into the CAS's 65_536 Target Socket Translation Table.
15:12	NPAST: Secure Network Engine (SNE) Index	This field contains the process-specific index of the socket's associated SNE. This field references an entry in the associated CAS's NPAST→NSAST, which in turn is used to extract a System SNE and a Socket Index mask for construction of the associated NSAST.
57:16	Bit 42 Must be Clear all Other Bits Set	This field must contain the value 0x3FFF_FBFF_FFFF for the associated reference to be interpreted as a Target NPR.

5.8.8 NPR-to-NSR Translation

A CAS contains a 16-entry Source PSNE-to-SSNE Translation Table (SPSNE→SSNE), a 16-entry Target PSNE-to-SSNE Translation Table (TPSNE→SSNE), a 65_536-entry Source Process Socket to Network Socket Translation Table (SPS→SS), and a 65_536-entry Target NPAST→NSAST Table, and a 65_536-entry Target Process Socket to Network Socket Translation Table (TPS→SS). The combined operation of an SPSNE→SSNE and an SPS→SS represents Source NPAST-to-NSAST translation (SNPAST→NSAST) and the combined operation of an SPSNE→SSNE and an SPS→SS represents Target NPAST-to-NSAST translation (TNPAST→NSAST). The SNPAST→NSAST and TNPAST→NSAST can be merged into a structure with shared and partitioned entries (a highly workable solution would use 24 entries: 8 source-specific, 8 target-specific, and 8 shared source-target entries)[1]. The SNE translation tables are small enough that the complexity of caching them in a PN transcends its potential value. Conversely, the socket tables are so big that they must be cached in a PN.

NPAST and Network Socket (NS) translations occur in parallel, with the former generating the NSR's NPAST (SNE Index and Socket Index) and the latter generating the NSR's Offset (either via substitution or addition). NPR translation is not associated with a cache lookup, in the traditional sense. It is used to generate a cache transaction, and as such, its latency is less critical (though the translation model is very fast).

1. This is an optimization that improves PN Table performance and is otherwise not particularly useful (CAS storage savings are so modest as to be considered a don't care). Conversely, less cache traffic and fewer translation misses can have a modest return for something that is essentially free..

Table 21: Process NPAST Index to System NPAST Index Translation Table Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
V/P	-		System SNE Index								Socket Index [Least Significant Set Bit Sized]																						

SNE = Secure Network Engine **V** = Invalid (0) / Valid (1)

Table 22: Network System Reference (NSR) Field Descriptions

Bit Range	Field Name	Field Description
19:0	Socket Index	This field contains a field whose less significant active bit count is determined by the bit position of the field's least significant set bit. This field contains an SNE Index extension capability. The NSR's Socket Index is formed by: (1) locating the position of the least significant set bit in this field, (2) constructing a 20-bit mask by setting all bits above the previous step's bit position and clearing all bits at and below the bit position (if the bit position is 6, then the mask would be 0xFFF800), (3) performing a bitwise-AND of the mask and the NPR's Socket Index and generating a Programming Exception if the result is non zero, (4) performing a bitwise-AND of the mask with this field and performing a bitwise-OR of the result with the NPR's Socket Index. The result of the last step in the generation algorithm is the translated NSR's Socket Index.

Table 22: Network System Reference (NSR) Field Descriptions

Bit Range	Field Name	Field Description
27:20	System SNE Index	This field contains the NSR translated SNE Index associated with the indicated NPR's SNE. This is a simple replacement of a 4-bit process-specific SNE Index with an 8-bit system-global SNE Index.
30:29	RESERVED	
31	Valid (P)	This flag indicates that the associated SNE translation entry is valid. When located in a CAS's SNE translation table and when clear, this flag indicates that any attempt to use the associated translation entry will result in the generation of a Protection Exception.

Table 23: Source Process Socket Format

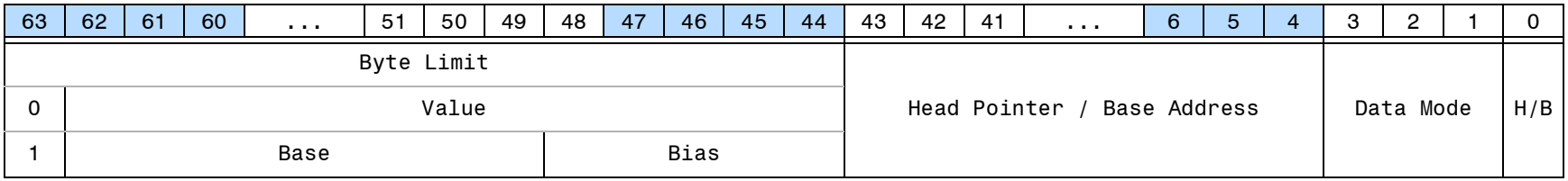


Table 24: Source Process Socket Field Descriptions

Bit Range	Field Name	Field Description
3	Head/Buffer (H/B)	This flag, when clear, indicates that the associated socket is configured for Head mode (all data is exclusively extracted from the head of the associated socket's allocation and the Head Pointer is automatically incremented by each byte read therefrom. The Offset field of NPRs in Head mode is reserved. When set, this flag indicates that the associated socket is configured for buffered operation and that the associated Head Pointer / Base Address field contains the associated socket's current Base Address.

Table 24: Source Process Socket Field Descriptions

Bit Range	Field Name	Field Description
0	Head/Buffer (H/B)	This flag, when clear, indicates that the associated socket is configured for Head mode (all data is exclusively extracted from the head of the associated socket's allocation and the Head Pointer is automatically incremented by each byte read therefrom. The Offset field of NPRs in Head mode is reserved. When set, this flag indicates that the associated socket is configured for buffered operation and that the associated Head Pointer / Base Address field contains the associated socket's current Base Address.
3:1	Data Mode	This field indicates that the associated socket's network traffic configuration. It is decoded as follows: • **0**: Invalid • **1**: A single byte data stream; independent of the associated write transaction size (an N-byte read is treated as N single byte reads); • **2**: A little-endian read of the associated data transaction size; • **3**: A big-endian read of the associated data transaction size; • **4**: RESERVED. • **5**: Big-endian, dual-byte data stream, independent of the associated read transaction size (an 8 byte read is treated as a quartet of 2-byte big-endian values). • **6**: Big-endian, quad-byte data stream, independent of the associated read transaction size (an 8 byte read is treated as a pair of 4-byte big-endian values). • **7**: Big-endian, octo-byte data stream, an 8 byte read is read most-significant byte first, least significant byte last
43:4	Base Address	The translated NSR's Offset field is the sum of the NPR's Offset and this field's value. The resultant location is not range checked by the associated NP's SAFE Adaptation Layer. SAFE's NAS model relies on its correct usage for proper operation. Only the more egregious errors are detected. SAFE prevents data leakage, but it will allow a process to move beyond its Tail Pointer.
63:44	Byte Limit	This field contains a standard Value(Base/Bias)(14,5) field that contains the maximum allowed count of bytes that are allowed to transit the associated socket. If more than this configured number of bytes are sent on the socket, then a Socket Overflow Exception is generated.This field's associated count value is calculated as follows, with "[N]" representing bit N of an translation entry and "[X:Y]" representing the bit range from bit X to Y of the entry: `([63]) ? [63:49] << ([48:44] + 14) : [63:44] << 8`, with 256-byte resolution (the smallest limit is 4096). A field value of 0 indicates that the associated Socket Index is invalid.

Table 25: Target Process Socket Format

63	62	61	...	46	45	44	43	42	41	...	6	5	4	3	2	1	0
Ring Counter							Tail Pointer							Data Mode			V/P

V/P = CAS [Invalid (0) / Valid (1)] / Cashe [Not Present (0) / Present (1)]

Table 26: Target Process Socket Field Descriptions

Bit Range	Field Name	Field Description
0	Valid / Present (V/P)	When located in a CAS's SNE translation table and when set, this flag indicates that the associated SNE translation entry is valid. When located in a CAS's SNE translation table and when clear, this flag indicates that any attempt to use the associated translation entry will result in the generation of a Protection Exception. When located in a PN's SNE translation cache and when set, this flag indicates that the associated SNE translation entry has been loaded from the associated process' CAS. When located in a PN's SNE translation cache and when clear, this flag indicates that the associated entry has not yet been fetched from the CAS table. Only valid SNE translations can be loaded into a cache. A cache miss associated with an invalid translation entry will trigger the generation of a Protection Exception.
3:1	Data Mode	This field indicates that the associated socket's network traffic configuration. It is decoded as follows: • **0**: Invalid • **1**: A single byte data stream; independent of the associated write transaction size (an N-byte write is treated as N single byte writes); • **2**: A little-endian write of the associated data transaction size; • **3**: A big-endian write of the associated data transaction size; • **4**: RESERVED. • **5**: Big-endian, dual-byte data stream, independent of the associated write transaction size (an 8 byte write is treated as a quartet of 2-byte big-endian values). • **6**: Big-endian, quad-byte data stream, independent of the associated write transaction size (an 8 byte write is treated as a pair of 4-byte big-endian values). • **7**: Big-endian, octo-byte data stream, an 8 byte write is written most-significant byte first, least significant byte

Table 26: Target Process Socket Field Descriptions

Bit Range	Field Name	Field Description
3	Head/Buffer (H/B)	This flag, when clear, indicates that the associated socket is configured for Head mode (all data is exclusively extracted from the head of the associated socket's allocation and the Head Pointer is automatically incremented by each byte read therefrom. The Offset field of NPRs in Head mode is reserved. The associated Head Pointer / Base Address field functions as the socket's Head Pointer. When set, this flag indicates that the associated socket is configured for buffered operation and that the associated Head Pointer / Base Address field contains the associated socket's current Base Address.
39:0	Tail Pointer	This field contains the associated socket's current Tail Pointer. This is the implied Offset of every write to the socket and it is incremented for every byte written.
63:40	Ring Counter	This field contains a count of the number of times the associated ring has wrapped. When combined with the Head Pointer / Base Address field, it becomes the count of the number of byte that have transited the associated socket.

5.9 Acceleration Address Space (AAS)

An accelerator is a processor that executes a standardized, shared processing capability based on Remote Procedure Calls (RPCs). The most commoditized instances of a Secure Acceleration Engine (SAE) is a Field Programmable Gate Array (FPGA) with a system controlled bitstream (the customized configuration (lookup-tables and associated routing) of an FPGA). A legacy CPU and GPU that runs a standardized graphics interface that interoperate with multiple concurrent clients would be another example. SAFE's SAE interface is Remote Procedure Call (RPC) centric, with a PN or System Component (SC) sending a message that contains an RPC to an SAE. An Acceleration Message (AM) invokes an RPC within its targeted SAE. For an SAE constructed from a CPU and GPU, this could be the invocation of a graphics privative that generates a bit-map that is either returned to the client or directly displayed on a monitor controlled by the targeted SAE.

SAFE's internal AAS data flow mechanism is physically a stream service that guarantees delivery or a complete AM and guarantees arrival order. SAFE's AASs are very similar to its Prioritized Notification Queue (PNQ) interface, with a larger message. A MN appends a 16-byte message to a queue associated with its targeted process. An Acceleration Message (AM) is 32 bytes in size, with the first 4 bytes identifying the origin of the AM, the next 4 bytes specifying the associated function and its argument configuration (termed a command), and the last 24 bytes carrying targeted function's argument list.

The 24-byte argument list is partitioned into arguments that are potentially as small a a single bit, but even with a very fine-grained partitioning mechanism, the supplied 24-bytes may not be sufficient for all of a function's requisite input arguments. The use of a shared DAS is the standard mechanism for expanding an SAE triggered RPC argument list and for passing data vectors between PN and SAE. For example, a graphics primitive can construct an arbitrarily large output bit-map and it may require and indeterminate number of input bit-maps to do so. These bit-maps will be shared via one of more DASs. An argument list that indicates a dozen input (size, location) tuples could be supported by an AM argument list that indicates a dozen arguments and the location of the dozen arguments tuples within a pre-arraigned DAS. This graphics centric SAE's DAS-based indirect argument list would then identify additional pre-shared DASs that contain the associated bit-maps (a single DAS could be used for everything, but static graphics bit-maps are more efficient

A process uses a Accelerator Process Reference (APR) to access a Accelerator Process Address Space (APAS), which is mapped to a Accelerator System Address Space (ASAS). An APR is used to generate an Accelerator System Reference (ASR) via the translation of Accelerator Process Address Space Tags (APASTs) to Accelerator System Address Space Tags (ASASTs) . The underlying accelerations that these ASTs represent support unicast data flows targeting an SAE and originating at either (1) a process; or (2) a System Component (SC), of which SAEs are a member. When an accelerator flow is associated with a process, it is tied to the process' CAS, not to the PN on which the process is executing. CAS association is required because: (1) the process may not be executing when an SAE executes the requested RPC, and (2) the process may not resume on the PN from which it requested the RPC.

SAEs employ pair-wise connections with clients. The connection is the mechanism used to map service requests (RPCs) to processes (CAS). A connection can have up to 2 DAS associations that can be employed for sideband communication of data blocks associated with RPC invocations. RPCs that require more than 2 DASs must chain connections (multiple connections at the SAE are associated with a single process connection). A connection is identified by an Acceleration Index.

ASASTs are SAE index routed. Each ASAST index is associated with a physical SAE and each physical SAE Index can have as many as 256*1024 user process Acceleration Indices and another 256*1024 Acceleration Indices that are dedicated to System Components (SCs). SCs rely on the system's inverse hardware to CAST mapping (Interrupt Capable Hardware Component (ICHC) have relatively small SC Indices are mapped to their associated CAST by a table in the associated

PMU). CH transactions identify the source of each transaction as part of the metadata wrapper within which each cache transaction is forwarded. An SAE has a table of source SC mappings against which it compares the CH's source indication and the associated cache transaction's source indication. All three must match to avoid the generation of a self-check of the sending SC.

Functionally, an SAE's SC ASR source checker, like many SAFE SC integrity checkers, represents the possibility that a rogue accelerator could make a system inoperable by consistently forcing a self-check of an LLS or PMU. The only defense against this is the system's logging infrastructure that identifies the source of self-check directives. The log is stored in an LLS (an SNE can be instructed to send a failure log) and a successful self-check followed by a subsequent self-check directive creates ambiguity within a system if it is a single component logically pointing its finger ant another component and no other components are in accord (the identity of the misbehaving entity is unclear). In general LLSs and PMUs are relatively more trusted components than PNs, SNE, and SAEs, but a system owner will ultimately need to intervene in a decision to functionally excise and/or replace an SC.

The failure of an SAE will trigger the failure of a processes that use the failed SAE (except those that actively monitor SAE state and short-circuit interaction with failing/failed SAE). SAFE does not acknowledge any distinction between interaction with a failed SC and one that never existed (Protection Exceptions are generated in response to both situations).

An ASR includes a 4-bit Accelerator Message Queue (AMQ) and a 6-bit Functional Partition. The AMQ identifies to which queue the associated AM is to be appended and the Functional Partition identifies a specific acceleration capability. AMQs can be extensions of the Functional Partition via AM function specific AMQ placement. A process' AMQ and Functional Partition selections are configured in the APR-to-ASR translation mechanism (not user selectable). An AM's Command Metadata field (the identification of the requested RPC service) is Functional Partition specific.

5.9.1 AAS Writes

When a process is writing to an AAS. The written data is appended to an AAS queue. A process cannot control the framing of an AM's creation (which takes 3 writes). A CAS contains a 2 word buffer that is used to construct an AM. The buffer holds the last two data words of an AM, and an AAS write constructs an AM when using the data written by the AAS write and the two words from the AM buffer. Although the AM buffer is in the CAS, the AM buffer is typically stored in the SAFE Adaptation Layer and written through to the CAS on a context switch.

5.9.2 AAS Reads

AASs are unidirectional. A process sends acceleration commands to SAEs. An SAE responds with an MN, when a response is indicated. There are no AAS reads (SAE AMQ reads ore internal to an SAE's implementation and not exposed to SAFE's infrastructure.

5.9.3 Accelerator Messaged Notification Model

AAS data flows unidirectionally from process to SAE. Acceleration Message (AM) handling is SAE implementation specific. A system selects an Acceleration Message Queue (AMQ) index for AMs that employ the associated process-to-system translation. AMQ delivery has an SAE implementation dependent notification mechanism. An SAE can request that a PMU temporary block a process that is overusing its queuing capabilities. SAE can request that an PMU withdraw a translation, Such a request includes the CAST of the process using the withdrawn translation.

5.9.4 Accelerator System Reference (ASR)

An Accelerator System Reference (ASR) is composed of a 28-bit Accelerator System Address Space Tag (ASAST) (8 bits of Secure Accelerator Engine (SAE) Index and 20 bits of Acceleration Index), an Accelerator Message Queue Index, and a 32-bit Command Metadata field..

Each SAFE process/thread can concurrently interact with up to 16 SAEs and up to 4096 acceleration connections per SAE (a total of 65_536 concurrent connections). A process that requires a higher accelerator interface count and/or acceleration count can be constructed from a vector of threads. A process that wishes to have more than 4096 concurrent acceleration interactions with a single SAE can configure the translation interface to employ a shared SAE top effect an extension of the Acceleration Index field.

The translation processes' Process SAE Index (PSAE) to physical/System SAE Index (SSAE) is controlled by a PSAE-to-SSAE translation table (`PSAE->SSAE`). As previously indicated, a `PSAE->SSAE` supports the blurring of the SAE Index / Acceleration Index distinction by allowing multiple PSAEs to reference the same SSAE, with an extended Acceleration Index. The expansion field for APRs is the SAE Index, allowing a process to expand it acceleration count via the use of additional SAEs.

An NP's SAFE Adaptation Layer includes a 16-entry `PSAE->SSAE`. and a vector of per Acceleration Index metadata that includes: the acceleration's the data model (base data unit width and endian-ness) and current Base Address. An APAS Reference (APR) represents a dynamically sized, sliding window into a ASAS.

Table 27: Accelerator System Reference (ASR) Format

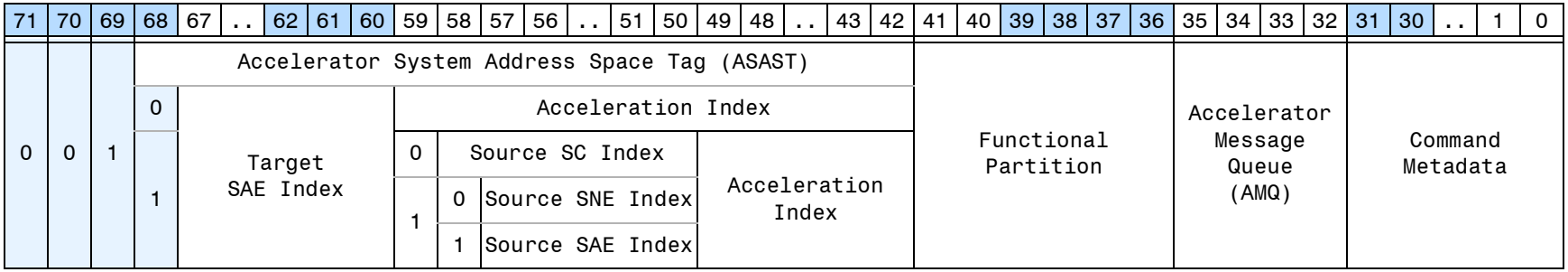


Table 28: Accelerator System Reference (ASR) Field Descriptions

Bit Range	Field Name	Field Description
31:0	Command Metadata	This field contains four bytes of Command Metadata that are delivered via address bits. An Accelerator Message (AM) is partitioned into fields based on this field's contents, which logically identifies a function call (RPC). This field's value, by identifying a remote function reference, indirectly indicates the partition boundaries within the associated AM's 24-byte Data Field field thar act as a arguments to that function. Software defined acceleration interlaces will frequently rely on one or more shared DASs and/or use SM to communicate large data, with the AM bytes providing size and offset tuple arguments into a shared data region..
35:32	Accelerator Message Queue (AMQ)	This field carries the queue into which the associated AM is to be placed at the targeted accelerator. The operation af queue at an accelerator (priority, physical partitioning/allocation of accelerator resources, and/or logical partitioning of accelerator functionality) is implementation defined. This field's value if defined by the APAST-to-ASAST translation. A 16-bit APAST value has an associated 4-bit AMQ value. An SAE has a maximum queue depth for each of its possible 16 AMQs. A process posting to an AMQ has no visibility into the associated queue's depth. An SAE can employ MNs or some form of shared memory to actively communicate an SAE's ability to absorb AMs.
41:36	Functional Partition	An SAE can support 64 concurrent acceleration profiles, though supporting one or two will probably be closer to the norm. An SC does not rely on APR to ASR translation and can hence dynamically select a target SAE's functional profile. A user process' APR-to-ASR translation mechanism supplies the Functional Partition for the ASRs it generates.
67:42	Accelerator System Address Space Tag (ASAST)	This field contains a system-wide identifier that contains the ASR's targeted 8-bit SAE and either a user process Acceleration Index of a System Component (SC) specific Acceleration Index. There are two major ASAST classes: user and system. User ASASTs are generated by APR-to-ASR translation. System ASASTs are directly specified by an SC. A user ASAST has n 18-bit Accelerator Index. A system ASAST has 3 subtypes: those sent by an SAE (ASAST contains an 8-bit source SAE Index and an 8-bit Accelerator Index); those sent by an SNE (ASAST contains an 8-bit source SNE Index and an 8-bit Accelerator Index); and those sent by an SC that is neither and SAE not an SNE (ASAST contains a 9-bit source SC identifier and an 8-bit Accelerator Index). An SAE can service 256*1024 concurrent processes and up to 256 acceleration contexts per SC.
59:42	ASAST: Acceleration Index	When ASAST bit 68 is clear, this field contains a user Acceleration Index (AI) with which the associated AM is associated. It identifies the user process requesting service. An SAE maintains an inverse mapping table from AI to CAST, allowing an SAE to respond to an AM with an MN.

Table 28: Accelerator System Reference (ASR) Field Descriptions

Bit Range	Field Name	Field Description
49:42	ASAST: Acceleration Index	When ASAST bit 68 is set, this field contains a system Acceleration Index (AI) with which the associated AM is associated. Unlike a user identifies the SC requesting service, an SC supplied AI is not necessarily associated at the target SAE with a source CAST. Source SAEs and SNEs are self identifying. Non SAE/SNE SCs employ an SC Source Mapping Table (up to 512 entries, though an SAE can support a smaller maximum non SAE/SNE SC count. A Source Mapping Table identifies the SC's CAST, allowing an SAE to respond to an AM with an MN.
58:50	Source System Component Index	When ASAST bis 68 is set and bit 59 is clear, this field contains a 9-bit SC Index. An SAE SC Index has an SAE specific inverse mapping that identifies the CAST associated with a SC Index, which allows the SAE to communicate with a requesting SC.
57:50	Source SNE Index	When ASAST bits 68 and 59 are set and bit 58 is clear, this field contains the SNE Index of the SNE associated with the RPC service request. For an SNE Index to be useful for inverse mapping, the SAE must have access to the system's SC to CAST mapping table (an SAE does not need to store the table, a small cache will typically be sufficient.
57:50	Source SAE Index	When ASAST bits 68, 59, and 58 are set, this field contains the SAE Index of the SAE associated with the RPC service request. For an SAE Index to be useful for inverse mapping, the SAE must have access to the system's SC to CAST mapping table (an SAE does not need to store the table, a small cache will typically be sufficient.
67:60	ASAST: SAE Index	This field contains the Secure Accelerator Engine (SAE) Index of the targeted accelerator. Acceleration Indices are allocated in 256*1024 blocks to SAEs, with the option of allocating multiple blocks to individual physical SAEs (thereby functionally extending the Acceleration Index into the SAE Index). SAE routing sends AMs to their indicated SAEs.
71:69	Must be 1	This field's value must be 1 for the associated SR to be interpreted as a ASR.

5.9.5 Accelerated Process Reference (APR)

Accelerated Process References (APRs). APRs are append only (write-only) addresses associated with 24 bytes of data. AMs are pushed at SAEs using Partial Cache Line Invalidation with Data CH transactions to get them to an L4 and Partial Cache Line Updates to forward them to their targeted SAE, without consuming anything more than short term buffering and bandwidth at the L4 (a multi-L4 system may involve an L4-to-L4 forwarding operation or the transaction transistions through an L5 and possibly an LLS before it can be sent back up the CH. AMs have a very short lives within their CH and SAEs must accept full responsibility for their buffering (the partial line push cannot be stalled or deferred).

Table 29: Accelerated Process Reference (APR) Format

57	56	. . .	50	49	48	47	46	45	44	43	42	41	. . .	34	33	32	31	30	29	. . .	2	1	0
1	1	1..1	1	1	0	Accelerated Process Address Space Tag (APAST)											Command Metadata						
						SAE Index				Acceleration Index													

SAE = Secure Accelerator Engine

Table 30: Accelerated Process Reference (APR) Field Descriptions

Bit Range	Field Name	Field Description
31:0	Command Metadata	This
47:32	Accelerated Process Address Space Tag (APAST)	This field contains an ASAST target-specific 16-bit index for the associated access. This is the process-specific identifier that is translated to an ASAST by the executing PN's hardware accelerated APAST-to-ASAST Translation Table and Cache, which in turn are dependent on the process' CAS-based APAST→ASAST. In general (SAE to SAE connections are also supported), one side of a accelerator connection (acceleration) is attached to a process and the other end is attached to a Secure Acceleratoring Engine (SAE). An SAE to SAE acceleration can more efficiently use DAS, MN, and SM infrastructures.
43:32	APAST: Acceleration Index	This field contains the process-specific index of the target Window on the associated SAE. This field references an entry in the associated CAS's APAST→ASAST, which in turn is used to extract a System SAE and a Window Index mask for construction of the associated ASAST.
47:44	APAST: Secure Accelerator Engine (SAE) Index	This field contains the process-specific index of the SAE on which the targeted Window is being managed. This field references an entry in the associated CAS's APAST→ASAST, which in turn is used to extract a System SAE and a Acceleration Index mask for construction of the associated ASAST.
57:48	Must be 0x3FE	This field must contain the value 1022 (0x3FE) for the associated reference to be interpreted as a APR.

5.9.6 APR-to-ASR Translation

A CAS contains a 16-entry PSAE-to-SSAE Translation Table (PSAE→SSAE) and a 65_536-entry Process Acceleration Index to System Acceleration Index Translation Table (PAI→SAI), The SAE translation table is small enough that the complexity of caching them in a PN transcends its potential value. Conversely, the acceleration table is so big that they must be cached in a PN.

PSAE→SSAE and PAI→SAI translations occur in parallel. APR translation is not associated with a cache lookup, in the traditional sense. It is used to generate a write transaction, and as such, its latency is less critical (though the translation model is very fast).

Table 31: Process SAE Index to System SAE Index Translation Table Format

63	...	38	37	36	35	34	33	32	31	30	29	28	27	26	...	21	20	19	18	17	16	...	4	3	2	1	0
-			Functional Partition						Accelerator Message Queue				System SAE Index					Acceleration Index [Least Significant Set Bit Sized]									V

SAE = Secure Accelerator Engine **V** = Invalid (0) / Valid (1)

Table 32: Accelerator System Reference (ASR) Field Descriptions

Bit Range	Field Name	Field Description
0	Valid (V)	This flag indicates that the associated SAE translation entry is valid. When located in a CAS's SAE translation table and when clear, this flag indicates that any attempt to use the associated translation entry will result in the generation of a Protection Exception.
19:1	Acceleration Index	This field contains a field whose less significant active bit count is determined by the bit position of the field's least significant set bit. This field contains an SAE Index extension capability. The ASR's Acceleration Index is formed by: (1) locating the position of the least significant set bit in this field, (2) constructing a 20-bit mask by setting all bits above the previous step's bit position and clearing all bits at and below the bit position (if the bit position is 6, then the mask would be 0xFFF800), (3) performing a bitwise-AND of the mask and the APR's Acceleration Index and generating a Programming Exception if the result is non zero, (4) performing a bitwise-AND of the mask with this field and performing a bitwise-OR of the result with the APR's Acceleration Index. The result of the last step in the generation algorithm is the translated ASR's Acceleration Index.
27:20	System SAE Index	This field contains the ASR translated SAE Index associated with the indicated APR's SAE. This is a simple replacement of a 4-bit process-specific SAE Index with an 8-bit system-global SAE Index.

Table 32: Accelerator System Reference (ASR) Field Descriptions

Bit Range	Field Name	Field Description
31:28	Accelerator Message Queue (AMQ)	This field contains the Accelerator Message Queue (AMQ) for all translations that resolve to the associated APAST→ASAST. This field is inserted into associated ASRs as part of the APR-to-ASR translation process, though this field is not the result of translation (it is just inserted).
37:32	Functional Partition	This field contains the Functional Partition for all translations that resolve to the associated APAST→ASAST. This field is inserted into associated ASRs as part of the APR-to-ASR translation process, though this field is not the result of translation (it is just inserted).

Table 33: Process Acceleration Index to System APAST Index Translation Table Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
V	-					Functional Partition						Accelerator Message Queue			

SAE = Secure Accelerator Engine **V** = Invalid (0) / Valid (1)

Table 34: Accelerator System Reference (ASR) Field Descriptions

Bit Range	Field Name	Field Description
3:0	Accelerator Message Queue	This field contains the Accelerator Message Queue (AMQ) for all translations that resolve to the associated APAST→ASAST. This field is inserted into associated ASRs as part of the APR-to-ASR translation process, though this field is not the result of translation (it is just inserted).
9:4	Functional Partition	This field contains the Functional Partition for all translations that resolve to the associated APAST→ASAST. This field is inserted into associated ASRs as part of the APR-to-ASR translation process, though this field is not the result of translation (it is just inserted).
14:10	RESERVED	
15	Valid (V)	This flag indicates that the associated SAE translation entry is valid. When located in a CAS's SAE translation table and when clear, this flag indicates that any attempt to use the associated translation entry will result in the generation of a Protection Exception.

5.9.6.1 Accelerator Message Format

Table 35: Acceleration Message Format

Word	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	..	41	40	39	38	..	33	32	31	30	..	25	24	23	22	..	17	16	15	14	..	9	8	7	6	..	1	0
0	-	SC	0	SC Index									Source Context Address Space Tag														Command																			
			1	0	SNE Index																																									
				1	SAE Index																																									
1	Data Byte 7								Data Byte 6								Data Byte 5								Data Byte 4								Data Byte 3								Data Byte 2								Data Byte 1								Data Byte 0							
2	Data Byte 15								Data Byte 14								Data Byte 13								Data Byte 12								Data Byte 11								Data Byte 10								Data Byte 9								Data Byte 8							
3	Data Byte 23								Data Byte 22								Data Byte 21								Data Byte 20								Data Byte 19								Data Byte 18								Data Byte 17								Data Byte 16							

SAE = Secure Acceleration Engine **SC** = System Component **SNE** = Secure Network Engine

Table 36: Acceleration Message: Word 0 Field Descriptions

Word / Bit Range	Field Name	Field Description
0 / 31:0	Command	This field contains an SAE implementation dependent 4-byte command field (an SAE has an interface specification that indicates the SAE specific encoding for this field). A Command indicates the RPC and the argument list format.
0 / 51:32	Source CAST	This field contains the CAST (process index) of the process that issued the associated MN. A Process can send a MN to itself, and if it does so, its CAST is used as the source, not the zero CAST self reference mechanism allowed by MN writes. When the an SNE, SAE, or other System Component (SC) is the origin of ther AM, the SC's CAST is determined by a table lookup performed on AM arrival (An SAE has a 20-bit, 1024 entry Inverse SC Mapping table that contains the CAST for every possible SC AM source.
0 / 60:52	System Component Index	When bits 63:62 contains 0b10, this field contains the configured System Component (SC) Index of the associated AM's sender. In general this field's value will be the same as the associated SC's CAST, except in cases where a system has more than 511, non-SAE/non-SNE Interrupt Capable Hardware Components (ICHCs)

Table 36: Acceleration Message: Word 0 Field Descriptions

Word / Bit Range	Field Name	Field Description
0 / 59:52	Secure Network Engine Index	When bits 63:61 contains 0b110, this field contains the Secure Network Engine (SNE) Index of the associated AM's sender. In general this field's value, plus 512 will be the same as the associated SNE's CAST, except in cases where a system has more than 511, non-SAE/non-SNE Interrupt Capable Hardware Components (ICHCs)
0 / 59:52	Secure Acceleration Engine Index	When bits 63:61 contains 0b111, this field contains the Secure Acceleration Engine (SAE) Index of the associated AM's sender. In general this field's value, plus 768 will be the same as the associated SNE's CAST, except in cases where a system has more than 511, non-SAE/non-SNE Interrupt Capable Hardware Components (ICHCs)
0 / 62	System Component	This flag, when set, indicates that the associated AM was sourced by a System Component (SC). When clear, the AM was generated by a user process.
0 / 63	RESERVED	
1 / 7:0	Byte 0	This field contains the byte 0 of the associated message's 24-byte data payload.
1 / 15:8	Byte 1	This field contains the byte 1 of the associated message's 24-byte data payload.
⋮	⋮	⋮
3 / 55:48	Byte 22	This field contains the byte 22 of the associated message's 24-byte data payload.
3 / 63:56	Byte 23	This field contains the byte 23 of the associated message's 24-byte data payload.

5.10 DSAST/NSAST/ASAST Cache Line Mapping

NSASTs and ASASTs are only addressed in caches. DSASTs have a relationship with a backing store in an LLS. Conversely, a PMU can store NSAST and ASAST content in its PMU specific DSAST (a PMU owns a 1pB address allocation in an LLS). A DSAST can be cache level limited to an L4 or L5, meaning that a cache line does not have an LLS-based backing store and it is not associated with a BAST (or GAST - no BAST means no GAST). A DSAST index has no relationship to its associated DPAST index, and if shared by multiple processes, it is likely that each sharing process will access it using a different DPAST. An NSAST index has no relationship to its associated NPAST index (source or target). An ASAST index has no relationship to its associated APAST index (source or target).

A SAFE system has multiple logically distinct cached ASs (DSAST, NSAST, ASAST, CAST (subsumes MNs), SM, and MSAST). The mechanism that is employed at a physical level is an implementation decision. An implementation can roll all the disparate ASs into a single unified caching infrastructure; an implementation can maintain independent, AS type differentiated cache infrastructures; a system can employ a small set of type specific cache lines and a larger shared pool of cache lines; a system can employ a partitioned CH with mutable partition boundaries (the relative allocation of DSAST cache lines can be increased at the expense of NSAST and/or ASAST cache lines); or some other model (limited only by the creativity of the associated implementation team).

Within any given cache, an AS is linearly mapped with a wrap from the largest to smallest allowed cache line index. DSAS mapping exists in a competitive mapping environment in which many DSASs can be mapped to the same limited number of cache locations. Due to size differences from cache to cache, each cache level has its own unique cache mapping, though within the generalized model. In systems that do not partition their caches, NSASs, ASASs, CASs, SM allocations, and MASs all compete for cache residence, with the caveat that any data that is not preserved n a lower level cache is not a candidate for replacement.

The presence of AS categories (DAS, NAS, CAS, etc.) represents a potential opportunity to expand L1 cache size and/or decrease L1 cache power by using AS category as an early

A par L1 SAFE cache uses

SAFE employs SAST index offsetting within its L1 and L2 caches: each SAST is offset by its SAST index within its associated caches (that is, the locations of SAST X, cache line y is (X + Y). (`DSAST[0].line[0]` aligns to physical cache line 0 (`PCL[0]`), `DSAST[1].line[0]` aligns to `PCL[1]`, `DSAST[4].line[6]` aligns to `PCL[10]`, `and DSAST[12].line[100]` aligns to `PCL[112]`. The offset prevents the alignment of SAST base addresses.

The Line Limit impact on the cache address is cache relative because of the impact of cache set size on Line Limits. The specified cache addressing model shifts the base of each successive SAST one line forward within the confines of the respective caches. Given the essential random nature of SAST assignments, this mechanism is as good as any until simulation based data improves on the general model (that improvement may be with regard to SAST assignment and not the their associated mapping).

5.11 InterProcess Communication (IPC) References

InterProcess Communication (IPC) References employ an independent trio of private address spaces that are cached like any/all other system addresses. The IPC addresses represent an address range, not a storage allocation. Both Shared Memory (SM) access and Messaged Notification (MN) access are common to the SRs and PRs. The third reference is a CAS reference at the system level and a SAFE State Register (SSR) reference at the process level. an MN and both of these third reference types are both CAS references, with the SSR a restricted form of CAS reference. SM has CAS-based state, but its data component is too large to fit in a CAS and it is allocated its own independent address allocation.

A CAS is maximally 24 bits (22 bits of which are allocated to MN buffers) in size. A maximally configured SM address space (20-bit Source CAST + 20-bit Target CAST + 24-bit Offset) represents a 16 exbibytes ($16*1024^6$ bytes), but an AS allocation, and an allocation whose associated memory/storage resource consumption is very closely tracked.

Table 37: InterProcess Communication System Reference Formats

<table>
<tr><th>71</th><th>70</th><th>69</th><th>68</th><th>67</th><th>66</th><th>65</th><th>64</th><th>63</th><th>62</th><th>...</th><th>45</th><th>44</th><th>43</th><th>42</th><th>...</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>...</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="3">1</td><td>S/U</td><td>0</td><td rowspan="3">0</td><td rowspan="3">0</td><td rowspan="3">0</td><td rowspan="3">0</td><td>H/L</td><td colspan="5" rowspan="3">Source CAST</td><td colspan="5" rowspan="3">Target CAST</td><td colspan="9">Messaged Notification Message</td></tr>
<tr><td>0</td><td rowspan="2">1</td><td>0</td><td colspan="9">Shared Memory Offset</td></tr>
<tr><td>1</td><td>1</td><td colspan="9">Context Address Space Offset</td></tr>
</table>

H/L = High(0) / Low(1) Prioritized Notification Queue **S/U** = System(0) / User(1) Prioritized Notification Queue

Table 38: InterProcess Communication Process Reference Formats

<table>
<tr><th>57</th><th>56</th><th>...</th><th>50</th><th>49</th><th>48</th><th>47</th><th>46</th><th>45</th><th>44</th><th>43</th><th>42</th><th>...</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>...</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="3">1</td><td rowspan="3">1</td><td rowspan="3">1..1</td><td rowspan="3">1</td><td rowspan="3">1</td><td rowspan="3">1</td><td rowspan="3">1</td><td rowspan="3">0</td><td>0</td><td>PNQ</td><td colspan="7" rowspan="3">Target CAST</td><td colspan="9">Messaged Notification Message</td></tr>
<tr><td rowspan="2">1</td><td>0</td><td colspan="9">Shared Memory Offset</td></tr>
<tr><td>1</td><td colspan="9">SAFE State Register (SSR) Index</td></tr>
</table>

PNQ = Prioritized Notification Queue

5.11.1 Messaged Notification Reference (MNR)

Messaged Notifications (MNs) unify hardware interrupts software signals behind a small message content (maximum 13 bytes) notification mechanism that supports communication between processes, between hardware components[1], and between processes and hardware components. Every Notification Capable Hardware Component (NCHC) is given a CAST, thereby logically becoming a process (software entity) as well as a hardware one. Every process is identified by its CAST. Using a CAST as the common thread between process and NCHC communication, MN allows any process or NCHC to interact via messaging with any other process or NCHC. NCHCs are allowed unrestricted access to other NCHCs and to processes.

An MN initiating process must know the target process' CAST to send it a message. Within a SAFE system a CAST is synonymous with a Process Identifier. Using an MN Process Reference (MNPR) an initiating/sending process' uses the target's CAST as part of the address, thereby routing the MN to its destination. The PN's SAFE Adaptation Interface translates the MNPR to an MN System Reference (MNSR), in part, by adding the senders CAST to the address, thereby identifying its source to its target. The sender indicates the priority of the message.

An inherently bi-directional, system global, Process Visibility Matrix (PVM) indicates whether IPC is allowed between an process pair (the matrix controls both MN delivery and Shared Memory (SM) usage). If a process attempts to notify a process index (CAST) that is not marked as visible in the PVM, a Protection Exception is generated. The PVM is a lower-left bit-map in which bi-directional communication is enabled by a set bit in `PVM[largerCAST,smallerCAST]` 2-dimensional array. The bits corresponding to the system's NCHCs are the only uni-directional PVM entries (an NCHC has unrestricted process access, but a process' access to an NCHC is restricted by the PVM). A clear bit in a PVM indicates that neither the associated CAST (processes) cannot communicate with each other. The PVM's upper-left-to-lower-right diagonal is not implemented because a process can always communicate with itself.

SAFE supports a maximum of four PNQ's per process/NCHC. There are high and low priority[2] system queues and high and low priority user queues. Each PNQ has its own independent configuration, with the scheduling of the associated process as a configuration's focus (in addition to delivering a small data content, notifications wake up processes). A notification capable entity can be constructed with all four queues or only system queues (any process that cannot generate a system notification will be unable to send a notification to such entities).

MNs unify interrupt and signals. They are also the primary structures to which SAFE's scheduling mechanisms are tied. For example, when a process is blocked, its time-out, wake-up event employs an MN to trigger the scheduling of the process'. SAFE scheduling (PMU operation) is inherently tied to a set of Time Rings and to the system's PNQs. PNQ depth, affected by message delivery, can move its associated processes from blocked to ready and/or escalate a process' priority within a ready queue.

There are three MN receipt modes:

- Unbuffered - The offset in associated MNPR reads is ignored and the next 8 bytes of the indicated PNQ's ring are returned, with associated increment of the ring's Head Index. If fewer than 8 bytes are

1. For purposes of MNs, hardware includes System Components (SCs) and other, less capable hardware (for example an L2 cache controller) of less sophistication. A Notification Capable Hardware Component (NCHC) that is not an SC has a limited repertoire, in terms of its notification processing capability. If an NCHC receives a message that it cannot parse, it sends a message to its PMU that indicates that it received such a message and it includes the sender and most of the message (the sender CAST replaces bits 127:108 of the message).

2. Relative PNQ priority is purely a configuration issue, with effective priority bound to the service urgency associated with a queue. They are termed high and low priority to express their intended, but the labels do not represent a prescriptive usage model.

read, data is lost (a single byte read will result in the discard of the upper 7 bytes of the associated message half's contents.
- Buffered - The MN queue is treated in much the same fashion as a buffered NPR. All references are relative to a queue's Base Address, a message can be read repeatedly, and the Base Address is explicitly advanced by the setting of the Advance Head (AH) flag in the reading MNPR.
- Null - MNs are disabled and all MN messages are promoted to the target NCHC's legacy interrupt or target process' legacy signal infrastructure (the NM's 24-bit metadata field is interpreted as an interrupt index when sent to a hardware process, or as a signal index when sent to a software process).

Each PNQ has a 16-bit Head Index and a 15-bit Tail Index. The tail references a message (the posting mechanism always appends a pair of 8-byte words). The Head Index is used to access a message 8-bytes at a time and hence it references half message boundaries. The indices can only be incremented and each wraps within its associated field width (each PMQ is a ring). The per PNQ index pairs are CAS-based and accessible via the process' SAFE State Register (SSR) interface. A PNQ can hold at most 32_767 messages (head == tail is an empty ring).

A process creates a notification by writing 8-bytes of data to an MNPR (if fewer bytes are written, the upper bytes of the message's data field will be zero). The MNPR (address) contains a target CAST, 3 bytes of data, and a target PNQ flag. A process is configured for system or user PNQ access, with a configuration enabled/disabled ability to control whether its MNPRs target system or user PNQs. When enabled, a process can change its target queue type via an SSR write. When disabled, a write to the SSR that modifies the process' current PNQ type will generate a Protection Exception. [Note: NCHCs live at the system level and do not use MNPRs to generate MNSRs.]

The PN's SAFE Adaptation Interface translates the MNPR to an MNSR by concatenating the source CAST, target CAST, and the 3 lower bytes of the MNPR into the MNSR's lower 64 bits and encoding the target PNQ type (system/user and high/low) and the MNSR code into the upper byte. The format sections below make this translation mechanism relatively obvious, with the exception of the appearance of an additional bit of PNQ type (supplied by the sending process' configuration).

The message creation mechanism places the written data into the second half of the message and places the lower 64 bits of the MNSR, with the target CAST replaced by the time at which the notification was enqueued, into the upper half of the message. The MN's target CAST and PNQ type are implied by the queue in which the message resides.

Sender's MNPRs only identify MN target CASTs (the source is implied). Target's MN message formats only identify MN source CASTs (the target is implied). MNSRs bridge the message arcs by explicitly identifying source and target CASTs.

The 11 bytes of data in an MN (3 bytes of metadata in first word and 8 bytes in the second word) are user defined. SAFE identifies standard models for extending the range of a message's delivery time, but these are user deployed, software models[1].

Once delivered, an MN's posted 16-byte data content can be retrieved by an asymmetric read address (an MN read uses a slightly different format from its write). Unbuffered PNQ reads return the next word (8 bytes) of a the targeted PNQ's ring. If the ring is empty, they return zero (not a valid value for a message, but rather a signal of an empty PNQ). A process can read its PNQ's status register (a read-only SSR) to determine a queue's current Head Index and Tail Index locations (subtracting the head from the tail (with wrap) calculates the number of unread messages).

Buffered PNQ reads are relative to PNQ's Head Index. A buffered read allows a process to select from within the queues current vector active messages and supports the rereading of messages from within a vector

1. There is a potential discontinuity between the time that hardware inserts into a message when it is enqueued and the time(s) that software inserts into a message when the message is created. Application software is responsible for managing this.

up to the point that the Head Index is advanced past a message. The MNPS's Offset field is added to the targeted PNQ's Head Index, with the result functioning as associated read's PNQ ring offset. The associate PMU manages a quartet (for entities that only support system ONQs, a pair) of 512kB rings within which the process' PNQs are managed. The PNQ rings are mapped into each processes CAS. Each 512kB ring can hold at most 32_767 MNs (an empty ring is indicated by the ring's associated Head and Tail indices being equal and a full ring is indicated by a Head Index that is one greater than its peer Tail Index, or the Head Index is 32_767 and the Tail Index is 0. A full ring buffer generates an Activity Exception (an exception that indicates that the associated process is being unresponsive to system processing requirements. Each PNQ has a High-Water Mark (HWM) and a Flood Mark (FM) that each affect the scheduling of the associated process. A HWM or an FM can message with a PMU to advance the scheduling priority of a process. The interaction between a process and it configured PNQ's HWM and FMs is detailed below within the PNQ Configuration section..

A buffered read includes a flag that optionally triggers the update of the PNQ's Head Index (the MNPR's Advance Head (AH) flag). Setting the AH flag advances the PNQ's Head Index to the specified Offset + 1:

```
Head_Index = (Head_Index + Offset + 2) & 0xFFFE.
```

A Head Index cannot be misaligned to a message boundary (its least significant bit muist be clear). As with an empty unbuffered PNQ, a read of an even invalid Offset returns a message with a Source CAST value of 0. A read of an odd Offset generates a Protection Exception (indiscriminate reading of a buffered PNQ is discouraged).

Buffered PNQ operation allows a process to scan its vector of a currently available message without copying them to alternative locations. It supports in-place message processing/handling. A process can read a message as many times as is useful for handling the message and/or the associated PNQ. A process cannot modify/update/alter the contents of a buffered PNQ. Once a PNQ's Head Index is advanced past a message, the message becomes inaccessible, independent of the PNQ's Tail Index location (a message would otherwise still be present within the buffer until overwritten by a subsequent message).

Notifications are how processes signal other processes that data is available (even if the available data is nothing more than the data within the notification itself); and notifications are how processes communicate with hardware, how hardware communicates with processes, and how hardware communicates with other hardware. There are other forms of IPC, but neither Shared Memory (SM) nor local or remote CAST access, via SSR, directly impact process execution. A PNQ's High Water Mark (HWM) configuration escalates process priority, potentially to the point of preempting execution in favor of a process with a HWM violating PNQ.

Any form of external control over the operation of a program can only take the form of polling of one form or another, that is, a program can only interact with externally supplied data by reading it. Even if an interrupt/signal triggers the execution of a handler or thread, the main flow of a program can only modify ir behavior by interrogating (polling) externally supplied data. Networking is the standard mechanism for passing data. Signals are the most common model for communicating more local and more process/thread centric information. A simple shim, in the form of an MN thread that waits on MNs can take the form of a signal handler and interrupt controller.

If a programmer wishes to dedicate a thread to "MN watching", that thread must explicitly block itself when there are no MNs for it to process (an empty queue does not constitute a blocking condition). Using a PMU to manage MNs makes it particularly easy to migrate MN watchers from waiting (blocked) to running because the PMU that manages MN receipt is also the PMU that transitions the process from blocked to running.Using this mechanism, transparency with legacy models is easily accomplished. If a PNQ's HWM is set to 1, a sleeping thread can be awakened by the delivery of a single message. An independent thread can be associated with each of a process' PNQs, allowing a process to extend its PNQs to a thread execution control mechanism.

SAFE's tying of its general, low-level messaging model to its CASTs does not expose the associated CAS's internals (process state) to its associated processes. Although the data being accessed is stores within a CAS, and although this might otherwise be problematic from a security perspective, there is no interface available to a process that can directly use a CAST index (the PNQs are indirectly accessed via a hardware controlled interface). CAST assignments are non-permanent (except for service processes which rely on well-known CAST indices). There is no relationship from execution instance to execution instance for the CAST(s) associated with a program and any of its peer programs (CAST associations are highly dynamic and it is highly unlikely that a process will be assigned the same CAST it had on a previous invocation if it executed again). Most importantly, IPC must be explicitly allowed via the Process Visibility Matrix (PVM). A process that attempts to spew MNs into a system will quickly kill itself. [Note: The use of CASTs to effect bi-directional communication obviates potential issues with asymmetric inverse message address mappings between processes.]

System attacks based on attempts to overwhelm a process'/NCHC's MN queue(s) with spurious messages and/or to force the creation of large blocks of SM data are possible (an internal form of a denial of service attack), are easily deflected. SAFE system's are advantaged by: a system level notification whenever a process creates a new L5 cache line, an inability to hide the source of MNs, and the ability to easily clear the flag that allows IPC access to a process/NCHC that being assaulted (thereby killing the attacker if it attempts another MN). It is also the case that the PMU(s) that mediates MN communication between attacker and target will quickly become aware of unusual MN traffic rates, potentially ahead of a process noticing them. A PMU can stall (unschedule.block) unvetted processes emitting significant numbers of MNs in anticipation of a process' complaint regarding the process.

Table 39: Messaged Notification System Reference (MNSR) Write Format

71	70	69	68	67	66	65	64	63	62	61	...	46	45	44	43	42	41	...	26	25	24	23	22	21	...	2	1	0
1	S/U	0	0	0	0	0	H/L	Source CAST							Target CAST							Metadata/Data						

H/L = High(0) / Low(1) **S/U** = System(0) / User(1)

Table 40: Messaged Notification Reference (MNR) Merged System/Process Write Field Descriptions

Bit Range	Field Name	Field Description
23:0	Data/Metadata	This field contains 3 bytes of Message Metadata or an additional 3 byte of Message Data. In general, this field contains the type of notification being delivered, but in an implementation/usage defined manner.
43:24	Target CAST	This field contains the CAST of the process to which the notification is to be delivered. Zero is not a valid CAST, however for a PR, a zero Target CAST field value indicates the issuing process' CAST is to be substituted in the generated MNSR as both the Target CAST and the Source CAST.
63:44	Source CAST	This field contains the source CAST (process identifier/index of sender) of the associated MN.

Table 40: Messaged Notification Reference (MNR) Merged System/Process Write Field Descriptions

Bit Range	Field Name	Field Description
64	High/Low	This field indicates whether associated notification it to be appended to one of the target's low priority queues (bit is set) or one of its high priority queues (bit is clear). Whether the system or user queue pair are being targeted is a process configuration artifact.
69:65	Must be 0	This field must be 0 for the associated reference to be interpreted as an MNSR.
70	System/User	This field indicates whether associated notification it to be appended to one of the target's user queues (bit is set) or one of its system queues (bit is clear). Whether a low or high priority queue is being targeted is indicated by bit 64.
71	Must be Set	This flag must be set for the associated reference to be interpreted as an MNSR.

Table 41: Messaged Notification Process Reference (MNPR) Write Formats

57	56	..	48	47	46	45	44	43	42	41	..	26	25	24	23	22	21	...	2	1	0
1	1	1..1	1	1	0	0	H/L	Target CAST							Metadata/Data						

H/L = High(0) / Low(1)

Table 42: Messaged Notification Reference (MNR) Merged System/Process Write Field Descriptions

Bit Range	Field Name	Field Description
23:0	Data/Metadata	This field contains 3 bytes of Message Metadata or an additional 3 byte of Message Data. In general, this field contains the type of notification being delivered, but in an implementation/usage defined manner.
43:24	Target CAST	This field contains the CAST of the process to which the notification is to be delivered. Zero is not a valid CAST, however for a PR, a zero Target CAST field value indicates the issuing process' CAST is to be substituted in the generated MNSR as both the Target CAST and the Source CAST.
44	High/Low	This field indicates whether associated notification it to be appended to one of the target's low priority queues (bit is set) or one of its high priority queues (bit is clear). Whether the system or user queue pair are being targeted is a process configuration artifact.
57:45	Must be 0x1FFC	This field must contain the value 8188 (0x1FFC) for the associated reference to be interpreted as an MNPR.

Table 43: Messaged Notification Process Reference (MNPR) Read Formats

57	56	..	48	47	46	45	44	43	42	..	19	18	17	16	15	14	13	12	..	4	3	2	1	0
1	1	1..1	1	1	0	0	H/L	-					S/U	Unbuffered Prioritized Notification Queue (UPNQ) Mode										
														-										
														Buffered Prioritized Notification Queue (BPNQ) Mode										
														AH	Offset									

AB = Advance Head **H/L** = High(0) / Low(1) **S/U** = System(0) / User(1)

Table 44: Messaged Notification Process Reference (MNPR) Read Field Descriptions

Bit Range	Field Name	Field Description
16:0	RESERVED (UPNQ Mode)	When the associated PNQ is configured for unbuffered operation, this field is reserved. An Unbuffered PNQ (UPNQ) simply returns the next word of the indicated PNQ's contents.
15:0	Offset (BPNQ Mode)	When the associated PNQ is configured for buffered operation, this field contains the Offset from the associated PNQ's current Base Address from which the associated reference pulls a data word (8 bytes). Even Offset values pull the first word of a message and odd Offsets pull the last word of a message. An MN read is always performed with a 8-byte aligned address (the lower 3 bits of this field are cleared) for internal processor addressing purposes (the lower bits will not trigger any unaligned address logic).
16	Advance Head (BPNQ Mode)	When the associated PNQ is configured for buffered operation, this flag, when clear, indicates that the position of the indicated MNQ's Head Index is not affected by the associated read operation. When set, this flag indicates that the MNQ's Head Index is set to reference the PNQ entry immediately following the entry referenced by thi reference. A Head Index can only reference the first word of a buffered message and the Head Index update operation indexes the next message independent of which message word is read by the associated reference.
17	System/User	This field indicates whether associated notification it to be read from one of the its user queues (bit is set) or one of its system queues (bit is clear). Whether a low or high priority queue is being targeted is indicated by bit 44.
43:18	RESERVED	
44	High/Low	This field indicates whether associated notification it to be read from one of the its low priority queues (bit is set) or one of its high priority queues (bit is clear).
57:45	Must be 0x1FFC	This field must contain the value 8188 (0x3FFC) for the associated reference to be interpreted as an MNPR read.

5.11.1.1 Messaged Notification Message Format

Table 45: Messaged Notification Format

Word	63	62	61	..	57	57	56	55	..	49	48	47	46	45	44	43	42	41	40	39	38	..	33	32	31	30	..	25	24	23	22	..	17	16	15	14	..	9	8	7	6	..	1	0
0	Arrival Time															Source Context Address Space Tag																		Meta Byte 2							Meta Byte 1							Meta Byte 0						
	0	Value																																										
	1	Base														Bias																													
1	Data Byte 7								Data Byte 6							Data Byte 5														Data Byte 4													Data Byte 3													Data Byte 2													Data Byte 1													Data Byte 0												

Table 46: Messaged Notification Field Descriptions

Word / Bit Range	Field Name	Field Description
0 / 7:0	Meta Byte 0	This field contains the first byte (byte 0) of the associated message's 3-byte metadata payload. Metadata can be purposed in any way the system designers deem appropriate, including the type of notification and/or reason for notification; or just 3 additional bytes of data.
0 / 15:8	Meta Byte 1	This field contains the byte 1 of the associated message's 3-byte metadata payload.
0 / 23:16	Meta Byte 2	This field contains the byte 2 of the associated message's 3-byte metadata payload.
0 / 45:24	Source CAST	This field contains the CAST (process index) of the process that issued the associated MN. A Process can send a MN to itself, and if it does so, its CAST is used as the source, not the zero CAST self reference mechanism allowed by writes. This means that the first word of a message will never be zero.

Table 46: Messaged Notification Field Descriptions

Word / Bit Range	Field Name	Field Description
0 / 63:46	Arrival Time	This field employs a standard Value/[Base/Bias](19/[15,4]) encoding to indicate the associated message's arrival time, with roughly 8ns resolution (the actual resolution is a values close to 8ns, for which multiples are easily generated using the associated system's clock). Two messages that arrive within the same ~8ns quantum have the same Arrival Time, but ordered within the associated queue based on their relative arrival time (a message that arrives 1ns before another message will be appended to the targeted PNQ first). This field's value is aligned to the system clock. A Value encoding will hold the least significant 19 bits of an effective ~8ns resolution system clock, with the possibility that the clock's lower 19 bits are about to wrap. The Value wraps on a somewhat more than 4ms quantum. If more range is required, the Base/Bias tuple expresses times from ~4ms to ~2.3 minutes, though with decreasing associated resolutions (~4ms to ~8ms resolution is 128ns, ~8ms to ~16ms resolution is 256ms, ~16ms to ~32ms.resolution is 512ms, and so forth until ~1 minute to ~2 minute resolution is ~4ms). Arrival times are relative to the system clock and users must adjust their indicated time accordingly (if a process executes every second and a n MN's Arrival Time is greater than the associated bits of the system time, the associated timer's wrap quantum has occurs at least once since the MN was posted). Messages that require more arrival time resolution can dedicate metadata and/or data bytes to extend a message's long-term resolution (a single byte could extend the effective wrap interval of a message's service delay to ~8.5 hours). Resolution extension is the responsibility of the message creator (it is not automatically supported by the system).
1 / 7:0	Byte 0	This field contains the byte 0 of the associated message's 8-byte data payload.
1 / 15:8	Byte 1	This field contains the byte 1 of the associated message's 8-byte data payload.
⋮	⋮	⋮
1 / 55:48	Byte 6	This field contains the byte 6 of the associated message's 8-byte data payload.
1 / 63:56	Byte 7	This field contains the byte 7 of the associated message's 8-byte data payload.

5.11.2 MN Queue Configuration

SAFE messages are an merger of an IPC mechanism with a scheduling mechanism. Messages, which merge signals and interrupts, have an obvious impact on scheduling. If a "cron[1]" mechanism is embedded into the MN system, then the PNQ configurations can act as the PMU's scheduling mechanism. A SAFE PMU runs a set of scheduling rings into which PNQ configuration attributes are placed that support self scheduling and PMU based process/thread time management via the associated process' PNQs.

Each process has a quartet of MN queues/rings. Hardware level queue handling is identical for all 4 queues and queue selection is determined by the process posting the message (a target cannot control a source's queue selection). Each Prioritized Notification Queue (PNQ) has a configuration that is represented by CAS data and an exposure of aspects of a PNQ's state is visible via the process' SSR interface. Processes interact with their PNQs in two major ways:

1. A process polls (occasionally reads) its PNQs and takes action when entries are found in a queue.
2. A process waits (stalls/blocks) for entries to arrive at a PNQ and message arrival triggers the transition of the process from blocked (waiting) to ready (eventually runnable). This model is frequently employed in a multi-threaded software environment, with a process, or group of processes sequestered to handle MNs and interact with other threads (processes) in the program using Shared Memory (SM).

An PNQ configuration includes:

- Coalescence Delay - Default behavior calls for the scheduling of a blocked process in response to the delivery of a message. This behavior can be delayed to pace notification delivery. A process can delay its transition to ready by a coalescence Delay setting.
- Wake Count - the number of entries in a PNQ required to override a process' Coalescence Delay. That is, a Coalescence Delay stalls overrides a process' transition to ready and a Wake Count overrides the Coalescence Delay's stall.
- Buffer Size - the maximum number of messages queue-able by the PNQ.
- Pending Message Count - the number of avtive messages in the associated PNQ.
- Head Index - the location of the PNQ's current base address.
- Tail Index - the location into which the next PNQ will be placed.
- High-Water and Flood Marks - the Pending Message Counts that will trigger an escalation of the associated process' priority.

PNQ configuration interacts with a process' Process Management register (CAS word) to trigger PMU mediated scheduling activities. A process runs on a schedule and optionally on-demand (via messages). When a process blocks, it is scheduled for execution at some future time, using its default execution quantum, if it has not already self scheduled using some other mechanism. Thus, every process that is blocked, has a scheduled point in the future that it will again be scheduled for execution, independent of any other external event (other than the passage of time). Once execution is scheduled, it cannot be unscheduled, but it can be configured to alert the PMU that handles its scheduling duties to skip an instance of its execution scheduling. A process that becomes runnable because of message arrival can indicate that the process has multiple time queue entries and that the next scheduled execution time should potentially be ignored, in favor of a later scheduled time created by an execution of the process that occurred before process' nearest scheduled execution time.

On-demand process execution schedules a process in response to the arrival of a message, potentially many messages, at one of its PNQs. Process scheduling is nominally triggered by the delivery of a single message, but scheduling can be delayed in the hope/expectation that

1. `cron` is the name for Unix's time based scheduling model. Its primary usage is associated with the configuration of a system to run specific processes at specific times, potentially on a repeated interval.

additional messages will arrive and they can be batch processed. Message delay can be overridden if the process' message count exceeds a pre-configured value. This means that a process can have its default execution scheduled and a post-message delayed execution scheduled, without the process being transitioned from blocked to ready.

A scheduling decision is made by a single PMU, creating a single point of execution for these actions and automatically resolving all parallel operational issues. Additionally, if scheduling becomes more complex than is trivially handles by a simple process scheduling mechanism, then the PMU can transition to a more complex scheduling mechanism (thereby increasing scheduling latency and power consumption) or recognize that the arrival of the occasional Spurious Activation (a process that runs ahead of its default schedule and that has nothing to do) MN is the result of an overly simple scheduling model (this is an unlikely occurrence because scheduling is relatively simple in a many-core system and a PMU can easily recognize most spurious activation conditions prior to creating on via the placement of a process in an execution queue ahead of its scheduled activation time and with empty PNQs.

A process can have a high scheduling priority, but a low execution priority (the must run at a time close to its scheduled time, but once its minimum execution time has been allotted, it has relatively low execution priority (a keyboard or mouse handler being a good example of such processes). A process can have low scheduling priority, but high execution priority (a background garbage collection process being a good example),

Within a SAFE system, application processors are never the targets of undifferentiated hardware interrupts. That is, unlike a legacy system in which an arbitrary process is swapped out of a PN so that the PN can be tasked with OS specific interrupt processing, a SAFE system's system components field all interrupts. A process interacts with the system via its Notification Queues and a process can poll its queues at its leisure (with the caveat that a process that violates MN responsiveness requirements is likely to be killed by its PMU). A PMU can preempt an running process in favor of a higher priority process. A PMU can actively support a co-operative threading model in which sets of processes and/or threads operate most effectively when scheduled for concurrent execution. The sophistication of a PMU's process/thread execution mechanism is advantaged by the availability of sequestered processing cycles within a PMU embedded system.

Table 47: Process Management: Configuration Word 0 Format

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	...	43	42	41	40	39	38	37	...	5	4	3	2	1	0
Scheduled Entry Count		Execution State		Sending PNQ Mode		Default Execution Priority			Default Scheduling Priority			-		Minimum Execution Quantum								Scheduled Time								
														Base					Bias			Base					Bais			

PNQ = Prioritized Notification Queue

Table 48: Process Management: Configuration Word 0 Field Descriptions

Bit Range	Field Name	Field Description
38:0	Scheduled Time	This field contains a standard Base/Bias (35,4) field that indicates a system time in quanta of ~8ns. The field's value generation formula is: `(~bias) ? (1,base) << bias : base`. It can schedule 9 minutes out with 8ns resolution. For each doubling of this time, the associated time resolution is halved, with a maximum time specification ~100 days into the future with 130us resolution. A process cannot be scheduled more than 100 days into the future (a scheduling process can be used to as a proxy for longer term scheduling). When the associated Execution State is Blocked, this field indicates the time at which the associated process is scheduled for execution (for a process with multiple concurrent scheduled times (the associated Scheduled Entry Count is 2 or 3). it indicates the time created by its most recent execution). When the associated Execution State is Ready, this field contains the time at which the process transitioned from Blocked to Ready. When the associated Execution State is Running, this field contains the time at which the associated process transitioned from Ready to Running. This field us updated by a process transitioning from Running to Ready or Blocked and so there is no interpretation conflict (a running process only sees its start time). This word is visible via a read-only SSR.
49:38	Minimum Execution Quantum	This field contains a standard Base/Bias (8,3) field that indicates the minimum amount of execution time a process is to allotted on a per scheduling instance (once it starts running, the minimum time it must remain in execution before it swapped-out). Time is specified in graduated 64ns quanta (64ns to 32us minimums with 64ns resolution. 16us to 32us minimums with 128ns resolution, and so forth until 1ms to 2ms minimums with 4us resolution). A field value of 7 indicates that there is no associated minimum execution time. A PMU will manage its PNs to ensure that one or more processes are always nearing their minimum execution time(s).
51:50	RESERVED	
54:52	Default Scheduling Priority	This field indicates the associated process' default scheduling priority (when a process transitions from blocked to ready, the relative advantage it is assigned for selection for execution (0 is the highest priority and 7 is the lowest priority). For example, when a PMU selects the next process to execute on a PN, it will choose a process with a scheduling priority of 2 over one with a scheduling priority of 4.

Table 48: Process Management: Configuration Word 0 Field Descriptions

Bit Range	Field Name	Field Description
57:55	Default Execution Priority	This field indicates the associated process' default execution priority (when a process is running, the relative advantage it is assigned for continued execution (0 is the highest priority and 7 is the lowest priority). For example, when a process with PMU has 2 processes that have both consumed their alloted minimum execution times are running, and one has a execution priority of 3 and the other has an execution priority of 6, and a the head of the ready process queue has an execution priority of 4, then the running process with a priority of 6 will be moved back onto the ready queue and the head of the ready queue will be transitioned to running. The ready queue is configured for both priority and fairness. The longer a process resides in the ready queue, the greater its instantaneous scheduling priority becomes (a PMU dynamically maintains an instantaneous per process scheduling priority that is independent of its default priority).
59:58	Sending PNQ Mode	A process can be configured with access to only its peer's user PNQs, only their system PNGs, or both. When a process can access both queue sets, it is configured for only one pair at a time. An SSR write is required to change a processes PNQ. **0** = System Only (the associated process only generates MNs that target system PNQs); **1** = User Only (the associated process only generates MNs that target user PNQs); **2** = Mutable, Currently System (the associated process can target user or system PNQ and is currently targeting system queues); and **3** = Mutable, Currently System (the associated process can target user or system PNQ and is currently targeting system queues)
61:60	Execution State (ES)	This field contains the associated process' current execution state: The encoding of this field is as follows: **0** = Blocked, **1** = Ready; **2** = Running (or in set-up: being prepared to execute but not yet executing); and **3** = In Tear-Down (no longer running, but within a PN's active state).
63:62	Scheduling Entry Count (SEC)	This field contains the number of scheduling entries associated with the process. Every process has at least one, except for a process that is no longer active (or that has been deactivated, which is independently tracked by the associated PMU). When this field's value is greater than 0, a triggered scheduling event, decrements this counter and, if the associated Scheduled Time is other than the current time, it discards the associated scheduling event (the associated process' state is unaffected by the event).

Table 49: Messaged Notification Queue Management: Configuration Word 0 Format

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	..	17	16	15	14	13	..	1	0
Mode		TBS		-		FMP			Flood Mark											HWMP			High-Water Mark											Base Address									Message Count					
									Base								Bias							Base								Bias														

FMP = Flood Mark Priority **HWMP** = High-Water Mark Priority **TBS** = Terminating Buffer Size

Table 50: Messaged Notification Queue Management: Configuration Word 0 Field Descriptions

Bit Range	Field Name	Field Description
14:0	Message Count	This field contains the number of active messages in the associated buffer. It is incremented whenever a message is added. A Processing Exception is generated if its value reaches 0x7FFF. Any messages that target s full PNQ are discarded.
29:15	Base Address	This field contains the Base Address of the associated PWQ (the current offset of the associated PNQ ring's Head Index). It is the location of the oldest valid message in the associated PNQ. This field wraps within a 15-bit reference space (the associated 15-bit value is a message Index (size-aligned 16-byte reference) not a byte reference) that corresponds to a location in a size aligned 512kB circular buffer. Its value is modified by Base Address updates and singleton MN reads.
40:30	High-Water Mark	This field contains a standard Base/Bias(8,3) encoding that indicates the Message Count value that triggers the escalation of associated process's scheduling priority from its current priority is to the priority indicated by the associated High-Water Mark Priority field. A field value at 0 disables this feature. Triggering of a process' High-Water Mark bypasses standard scheduling mechanism and places the associated process directly into the indicated execution priority queue. The High-Water Mark formula for the field is: `(~bias) ? (1,base) << bias : base`. Its maximum value is 32_704 (a value perilously close to getting the process killed by overflowing one of its PNQs, but the associated bias starts at 16_384, a potentially sane High-Water Mark value).

Table 50: Messaged Notification Queue Management: Configuration Word 0 Field Descriptions

Bit Range	Field Name	Field Description
43:41	High-Water Mark Priority (HWMP)	This field contains a the value to which the associated process' Execution Priority is to be elevated if the associated Message Count field's value becomes greater than or equal to the configured High-Water Mark. If the associated process is blocked, it schedules the process at the associated priority. The High-Water Mark triggering test is made when the Message Count is incremented and it is triggered when the count become equal to or larger than High-Water Mark field's value.
54:44	Flood Mark	This field contains a standard Base/Bias(8,3) encoding that indicates the Message Count value that triggers the escalation of associated process's scheduling priority from its current priority is to the priority indicated by the associated Flood Mark Priority field. A field value at 0 disables this feature. Triggering of a process' Flood Mark bypasses standard scheduling mechanism and places the associated process directly into the indicated execution priority queue. The Flood Mark formula for the field is: `(~bias) ? (1,base) << bias : base`. Its maximum value is 32_704 (a value perilously close to getting the process killed by overflowing one of its PNQs, but the associated bias starts at 16_384, a potentially sane Flood Mark value).
57:55	Flood Mark Priority (FMP)	This field contains a the value to which the associated process' Execution Priority is to be elevated if the associated Message Count field's value becomes greater than or equal to the configured Flood Mark. If the associated process is blocked, it schedules the process at the associated priority. The Flood Mark triggering test is made when the Message Count is incremented and it is triggered when the count become equal to or larger than Flood Mark field's value.
59:58	RESERVED	
61:60	Terminating Buffer Size (TBS):	This field contains a PNQ's Message Count range, with values 0 through 3 representing a maximum range of 0:4095, 0:8191, 0:32_767, and 0:65_535, respectively. Range restrictions are considered to represent a physical limitation, independent of the backing implementation. A violation of the associated PNQ's physical ring management structure will cause process termination. A process that attempts to kill another process by overflowing one or more of its PNQs will be blocked by the associated PMU, the targeted process will be started, and the targeted process will indicate to the PMU whether shenanigans are at work (in which case the PMU will kill the shenaniganist).

Table 50: Messaged Notification Queue Management: Configuration Word 0 Field Descriptions

Bit Range	Field Name	Field Description
63:62	Mode	A field value of **0** indicates that user control of the associated queues Base Offset is disabled and that the associated PNQ operates in singleton mode, meaning that the read address offset is ignored when accessing an MN queue. A field value of **1** indicates that the enhanced sliding window capability is enabled. A field value of **2** Reserved. A field value of **3** indicates that the associated PNQ is disabled and triggers the delivery of a message to the PMU and the associated messages origin, indicating use of a disabled resource. Generally, the PMU message will trigger the PMU's termination of one or both of the associated processes.

Table 51: Messaged Notification Queue Management: Configuration Word 1 Format

63	62	61	...	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	...	30	29	28	27	26	25	24	23	22	21	...	7	6	5	4	3	2	1	0
-							Wait Override											Coalescence Quantum											Default Quantum											
							Base								Bias			Base								Bias			Base								Bias			

Table 52: Messaged Notification Queue Management: Configuration Word 1 Field Descriptions

Bit Range	Field Name	Field Description
23:0	Default Quantum	This field contains a standard Base/Bias(19,5) field that indicates the number of ~8 nanosecond quanta that the associated process is to wait before being sent an automatically generated Timeout MN. A field value of 0 indicates that the Default Quantum timer function is disabled. The maximum quantum for the associated field is (0xFFFFF << 30)*8ns (a bit more than 104 days, far larger than is otherwise useful, but field sizing was aimed at getting precision up - when scheduling range is an hours, scheduling accuracy is a milliseconds). Timeout messages bypass a process' message coalescence . The quantum generation formula for the field is: `(~bias) ? (1,base) << bias : base`

Table 52: Messaged Notification Queue Management: Configuration Word 1 Field Descriptions

Bit Range	Field Name	Field Description
39:24	Coalescence Quantum	This field contains a standard Base/Bias(12,4) field that indicates the number of ~8 nanosecond quanta by which the associated process' shroud delay its transition from blocked is be ready, following the arrival of the first, post-execution MN (the first MN that arrives following its transition from running to blocked). A field value of 0 indicates that the Coalescence Timer function is disabled. The maximum quantum for the associated field is (8192 << 14) * 8ns (a bit more than a second). The quantum generation formula for the field is: `(~bias) ? (1,base) << bias : base`
48:40	Wait Override	This field contains a standard Base/Bias(8,3) field that indicates the depth of the associated PNQ that will trigger the transition of the associated process to ready, independent of the associated Coalescence Quantum. The associated PMU does not make a scheduling entry when this occurs, instead it places the associated process directly into an execution queue. A field value of 0 indicates that the override function is disabled.The MN override count formula for the field is: `(~bias) ? (1,base) << bias : base`. The maximum override value is 32_704 (a value perilously close to getting the process killed by overflowing one of its PNQs, but the associated bias starts at 16_384, a potentially sane override value).
63:49	RESERVED	

5.11.2.1 MN-Assisted Socket Messaging

SAFE supports three interlocked and/or independent mechanisms for supporting communication between processes (within a SAFE environment, communication between a local process and a remote process via an SNE is functionally communication between a process and an SNE that acts as a proxy for a remote system communications). Small scale communication (e.g. keyboard, telnet, telephony) can be exclusively supported by embedding the data to be moved directly into a MN (up to 11 bytes of data can be communicated per MN, though at least 1 of those bytes is generally required to identify the type of the associated message).

Higher bandwidth communication within a purely legacy socket model can communicate via a combination of an SM data channel and MNs that signal the presence and meta content of the data within the SM channel. Simpler SAFE implementations need not support Network Address Spaces (NASs). Basic intra-system networking can easily be handled by SAFE's MN and SM mechanisms; and low bandwidth external communication can also be handled by a very simple (restricted capability) SNE.

High-bandwidth communication and/or many-socket communication can employ a usage optimized combination of NASs, SM, and MNs to move data, with the default mechanism using MNs and NASs. The use of any given data transport model does not preclude the use of other transport models. As can be seen, all 3 basic transports: (1) MN only,

(2) SM + MN, and (3) NAS + MN; rely on MNs, though an SM or NAS only model could be constructed that relies exclusively on polling (in place of an interrupt driven MN interface).

When constructing a NAS based system, MN messages are generally used to communicate NAS state between sender and receiver. NAS writes do not have an associated NAS specific notification mechanism and they rely on the MN model for notifications[1]. By default, an NAS writer (effected via an NPAST mapping to the NSAST) sends an NAS related MN in association with the data being transitionally posted to its target NPAST. The sender posts the position of the NPAS's tail as the data word of its MN (the socket's current tail Index location is also available as an SSR value, but an SSR update does not wake the associated process).

A single byte can trigger an MN, but it is not the byte being posted to the NPAS that generates the MN. The MN is associated with the update of the NPAS's tail Index in the process' CAS. Tail Index updates tend to be data block oriented (the Tail Index is not updated for each byte posted to the NSAS). MNs are sent based on a simple rule set and based on the Tail Index's value (a source can post many bytes to an NSAS before it updates its tail Index, and once an MN has been posted, additional MNs are disabled until they are explicitly enabled via a write to the associated queue's MN enable SSR).

A data target dynamically reads its NPAS's Tail Index (via an SSR) when processing received data. The contents of an MN's may not accurately represent the associated NPAS's current state (the first notification can prevent a significant number of subsequent notifications). After the target processes a NPAS, based on the receipt of an MN, it sends an acknowledgment MN that includes the target's head Index, representing the amount of data the receiver has recognized as being transmitted. The acknowledgment is independent of the associated NPAS's buffer management (the exclusive purview of the explicit Advance Base flag, Base Address centric buffer management model) NPAS source MN usage is disabled until the source receives an acknowledgment from the target. The communicated target's head index and source's tail index are used as a critical component of MN notification rules.

MN NPAS messaging configuration is associated with the NPAS source, the dominant characteristics being the source's Tail Index, the target's Head Index, a byte count, and a delay. A delay of 0 and a byte count of 1 will result in a MN being generated as soon as the associated NSAST's tail Index is updated. A delay of 500ns and a byte count of 4096 will result in a MN being generated as soon as the associated NSAST's Tail Index has moved 4096 bytes or as soon as it has moved at least 1 byte and 500ns have elapsed since that first byte of data arrived (an MN is generated if: after 500ns if 1 byte is received and no more data arrives within the subsequent 500ns; or 4096 bytes arrive independent of the time required to enqueue the bytes (on a 100gb Ethernet interface 4096 bytes can be enqueued in less than 400ns)). A count of 0 disables data arrival triggered MNs for the associated SAST.

Virtually multicast NASs (a NAS whose data stream is made avialable to multiple reading processes) rely on the multicast proxy to generate MNs to the multicast peers to keep them appraised of the state of the data stream.

1. SAFE's process management model is triggered by MNs. A process that is blocked on a NPAST will only unblock when an MN triggers the associated process. This mechanism allows interrupt/notification/signal suppression to function as intended, with an MN arriving when the associated process is to be activated.

5.11.3 Shared Memory Reference (SMR)

SAFE's Shared Memory (SM) facility is a simple, no configuration, single writer, single reader data transport that allows one process to write data into an extra-CAS SM partition (the shared memory region is an enormous address space that significantly transcends the system's CAS mapping, but the SM is logically an extension of the CAS, hence, extra-CAS). A writing process has a vector of independent, per peer (CAST indexed) ASs. The maximum size of a CAST-to-CAST SM partition is 16mB (it has a 24-bit offset). Per CAST partitions can be uniformly configured to a smaller size (every partition is the same size system wide). The partitions are address space not memory/storage and unused address space has no system cost. A SAFE system can limit its per process SM buffer size to powers-of-2 corresponding to bit counts of 14 to 24 (16kB to 16mB).

The system's Process Visibility Matrix (PVM) indicates which of a process' SM partitions are accessible by their associated process. The SM space for a process is a 16tB byte vector (44-bit address) in which each 16mB block (the lower 24 bits of the address) is associated with the CAST identified by the upper 20 bits of the 44-bit address. A process can write anywhere in this 16tB block, but it cannot read it back. An SM buffer is uni-directional, one reader and one writer (it is not useful as local process storage and writing data to an buffer associated with an invisible CAST is highly questionable, though not prohibited. A process' entire 16tB address space is marked as invalid and flushed from the cache when a process is terminated.

The per process 16tB blocks are ordered by CAST index into a 16eB (16 exbibyte) vector (1024^2 CASTs x 1024^2 CASTs per CAST x 16gB per CAST), which at the SR level is a 64-bit field. This 16eB address space is a permanent system space and CASTs cannot be reused before a previous CAST instance's 16tB SM region has been scrubbed from the CH.

A CAST with visibility to another CAST can read its SM partition at the visible cache. A process read that would otherwise create a cache line (the line would only be created if it has not yet been written) kills the process attempting its creation with a Protection Exception. If the cache line exists because it has been written by the peer process, or because it is in the nulled region of a cache line that has been created by its writing process, then the resident data is read (logically null cache line content reads as zeros).

The maximum size of a configured SM range does not impact the associated system address generation. Instances of system addresses in limited process count (a logically smaller than 20-bit CAST index - fewer than 1024^2-1 processes) and/or limited offset size scenarios peer buffer configured for as size smaller than 16mB), have leading zeros in their CASTs and/or Offset fields (it is a 64-bit field with a pair of 20 bits CAST indices and a 24-bit offset independent of and physical or configured limitations. The SM allocation is generally very sparse matrix (significant holes exist between their active/valid regions and the empty space is easily compressed by the LLS associated with the SM's backing store).

An SM Writer places data into a system address space owned by the writing process (CAS) and within that allocation, into buffers associated with (accessible by) the indicated reading CAS (Process). Address space allocation errors are associated with the writer (a remote process cannot generate a local process error). An SM reader (target) pulls data from the locations previously written by its source. Process 'A' sends data to Process 'B' by placing the data to be communicated within its 'B' specific buffer, from which 'B' can read it at its leisure. If 'A' needs data back from 'B', then 'B' places the associated data into its 'A' specific buffer, from which 'A' can read it.

SM writes and reads, like standard memory operations, have no associated notification or transaction logging. Readers must either poll for buffer updates (a well-known offset must ne used and initialized to prevent a process killing attempted reader cache line allocation), or rely on MNs to create a poll free SM data movement model. SM buffers are memory. SM based communication requires a software maintained protocol to work effectively. SAFE does not specify that protocol and the

pair-wise buffer sharing mechanism doe not require that whatever protocol is being employed be universally applied (multiple, domain specific protocols can be employed above SAFE's guarantees of SM buffer integrity).

Table 53: Shared Memory Allocated Buffer Regions

Writing CAST	Reading CAST	X Byte Buffers				
		X-1	X-2	..	1	0
0	0	Writing-CAST[0]/Reading-CAST[0] Buffer				
	1	Writing-CAST[0]/Reading-CAST[1] Buffer				
	:	:				
	N	Writing-CAST[0]/Reading-CAST[N] Buffer				
1	0	Writing-CAST[1]/Reading-CAST[0] Buffer				
	1	Writing-CAST[1]/Reading-CAST[1] Buffer				
	:	:				
	N	Writing-CAST[1]/Reading-CAST[N] Buffer				
:	:	:				
N	0	Writing-CAST[N]/Reading-CAST[0] Buffer				
	1	Writing-CAST[N]/Reading-CAST[1] Buffer				
	:	:				
	N	Writing-CAST[N]/Reading-CAST[N] Buffer				

Table 54: Shared Memory System Reference (SMSR) Format

71	70	69	68	67	66	65	64	63	62	61	...	46	45	44	43	42	41	...	26	25	24	23	22	21	...	2	1	0
1	0	1	0	0	0	0	0	Source CAST							Target CAST							Shared Memory Offset						

CAST = Context Address Space Tag

Table 55: Shared Memory Systems Reference (SMSR) Field Descriptions

Bit Range	Field Name	Field Description
23:0	Offset	This 24-bit field contains the Offset in the writer owned SM region of the associated memory transaction.
43:24	Target CAST	When associated with a read, this field contains the CAST to be read. When associated with a write, this field indicates the CAST into which data is to be placed.
63:44	Source CAST	This field contains the origin of the associated transaction.
71:64	Must be 0xA0	This field must contain the value 192 (0xA0) for the associated address to be interpreted as an SMSR.

Table 56: Shared Memory Process Reference (SMPR) Formats

57	56	..	49	48	47	46	45	44	43	42	41	..	26	25	24	23	22	21	..	2	1	0
1	1	1..1	1	1	1	0	1	0	Target CAST							Shared Memory Offset						

Table 57: Shared Memory Merged (System/Process Reference) (SMSR) Field Descriptions

Bit Range	Field Name	Field Description
23:0	Offset	This 24-bit field contains the Offset in the writer owned SM region of the associated memory transaction.
43:24	Target CAST	When associated with a read, this field contains the CAST to be read. When associated with a write, this field indicates the CAST into which data is to be placed.
57:44	Must be 0x3FFA	This field must contain the value 14_204 (0x3FFA) for the associated reference to be interpreted as an SMPR.

5.11.4 Context Address Space Reference (CASR)

Forward looking SAFE based systems employ a hardware based process management and context switching model. A CAST is synonymous with a legacy system's process identifier/index and its referenced CAS subsumes the notion of a `proc` structure. A CAS references an abstract mapping of process state that is sufficiently sophisticated to support any processing context. Within a CASs structures like NPAST-to-NSAST and DPAST-to-DSAST translation tables are fully enumerated tables (unlike their associated PM based partitioned table/cache model). A CAS includes a multi-megabyte region at its end that can be used to hold any PN state that is not anticipated by the standard CAS structure.

A CAS is contained by a 16mB of address space. The associated storage allocated within the address range is process and implementation specific. A system's CAS Vector is a linear contiguous vector of 1024^2 CASs, one per possible process. This is logically a 1024^2 element array indexed by 20-bit CASTs in which each element is a 16mB (24-bit) address space. This is the generally sparse address space associated with CASRs. Although a CAS Vector belongs to the PMU that controls the associated processes, that PMU will typically place its CAS vector in an LLS backed DAS. Although a PMU's CAS vector, or portions thereof, are stored in a DAS, the associated LLS cannot typically view its plain/clear text contents (due to PMU specific encryption). SAFE system functional partitioning is based on a fundamental lack of trust, including peer System Components (SCs), for example PMU and LLS.

A process with visibility to another CAST can read its CAS partition via SSRRs. A process can never read a CAS, its own or another process' except via the SSRR interface and unless it has visibility to the other process. A process can always read its own SSRs.

All CASs are owned by their managing PMU and that PMU either directly or indirectly manages all changes to its CASs. Automatic process management within a PN is largely concerned with acquiring, caching, and then discarding the CAS state of the process a PMN is executing, and then doing the same for the state of the next CAS to be executed. Simple DMA capability can substantially reduce the processing overhead associated with context switching.

Processor state is the purview of a PMU's Processor State Management Engines (PSMEs). A PSME pulls PN process state from a recently blocked process back into its CAS and pushes the CAS state of the next process to execute into a PN. Unlike a normal processor, a fully SAFE aware processor does not load the entire hardware state of a process before its triggers the execution of the loaded process. Only small amounts of state are necessary for a process to begin execution. A PN's process state is partitioned into registers (must be present with a PN for the associated process to begin execution), tables (state that need not be present when a process begins execution, but that will be pushed into the PN's state while the process executes), and caches (state that is demand fetched on an as needed basis). A request for table data that has not yet been pushed into a PN's state can trigger an out-of-order PN state pull, or the PSME directed push to occur. The partitioning of PN state into its 3 components will be informed by implementation.

At some point CAS state moves into a PN. From a management perspective, CAS state lies on a usage dependent continuum between fully-static and highly dynamic. Rather that attempting to dynamically respond to CAS state update rates, appropriately sized PN caches can be used to manage PN based context. A PMU's PSME knows what needs to move towards a process to get it runnable, but outbound PN state associated with a process being swapped out req'suires that a PMU fetch a dirty state map from a PN to control its PSME operation[1].

As previously indicated, CAS state is owned by its associated PMU. That PMU can update their CAS state without restriction. Other SCs,

1. The use of relatively lower cost, but equally performant PMU based DMA engines, offloads simple data movement tasks from relatively more expensive application processors, which if multi-threaded, can be more efficiently employed executing applications that swapping contexts.

including other PMUs are allowed restricted access to the state with which they directly interact and all other access is restricted, just like any other process[1]. To affect CAS state beyond its allowed partition, a request must be made to its owning PMU.

Relative to its PMU, a CAS is just memory. CAS coherency is managed by an invalidation of cached data state prior to the manipulation of its PMU private/local state (CAS updates do not represent a race condition, each word of CAS has an exclusive update agent[2]). A modification of CAS state that is read-only at a PN requires that the PMU invalidate, lock (stall a request from a higher level cache), modify, and unlock the CAS state, which then enters the block/push/pull state that data had when the process are being loaded for execution.

State like a process' PNQ Message Count is expressed by the SAFE State Register (SSR) interface and their CAS state is represented by a combination of CAS state and software state. Conversely, state like a process' PNQ Base Address is not directly mutable by software, but it is indirectly modified by usage and control of the field is transferred to a PN's SAFE Adaptation Layer, with the requirement that an update of the PMU's CAS must be performed in association with a context switch.

CAS writes and reads, like standard memory operations, have no associated notification or transaction logging. Conversely, unlike normal memory, CAS content can be tied to hardware operation and hence changes to a CAS can significantly affect low-level system operation. The CAS memory map is detailed in Appendix B.

Table 58: Context Address Space Buffer Vector

16*1024 * 1024-1	16*1024 * 1024-2	...	2	1	0
CAST[0]					
CAST[1]					
CAST[2]					
:					
CAST[1024* 1024 - 2]					
CAST[024* 1024 - 1]					

1. An attempt by a system component to access restricted data is highly problematic, from a system perspective. In general, it indicates that the accessing system component has crashed (is no longer function properly), but still active. It may indicate that a system component has been compromised (the challenge of compromising a system component is significant, particularly given the isolation in which it operates, but the possibility must be addressed). The normal response to erroneous system component behavior is to message all NCHC nodes, record the identity of the offender, allow all peer components to save what they can, and reset the system.
2. For some words of a CAS, the identity of the update agent is state based, with control transitioning to delegated hardware state machines during execution and then being withdrawn back to the PMU when a process is not executing.

Table 59: Context Address Space Reference (CASR) Format

71	70	69	68	67	66	65	64	63	62	61	...	46	45	44	43	42	41	...	26	25	24	23	22	21	...	2	1	0
1	1	1	0	0	0	0	0	Source CAST							Target CAST							Context Address Space Offset						

Table 60: Context Address Space Reference (CASR) Field Descriptions

Bit Range	Field Name	Field Description
23:0	Offset	This 24-bit field contains the Offset within the CAST region that is target of the associated memory transaction.
43:24	Target Context Address Space Tag	When associated with a read, this field indicates the CAS from which the read will extract data. When associated with a write, this field indicates the CAS into which the write will place data.
63:44	Source CAST	This field contains the origin of the associated read/load or write/store transaction.
71:64	Must be 0xE0	This field must contain the value 224 (0xE0) for the associated address to be interpreted as an CASR.

5.11.5 SAFE State Register Reference (SSRR)

SAFE supports an access restricted abstraction of its processes' contexts via its SAFE State Registers (SSRs): A process cannot directly access its own Context Address Space (CAS), nor the CASs of relevant peer processes. However, a suite of important run-time CAS-based state information is made available to a process through its SSR vector. This allows a process to, for example, read the current depth of one of its Prioritized Notification Queue (PNQ). This interface exposes the set of readable, writable, and read/writable system state required to allow processes to securely execute many of their system calls via user space adaptations that are both significantly faster and lower overhead that those of legacy systems.

SSRRs are included in the set of IPC ASs, because the state of other processes can be interrogated via this interface, even though it is more common that these registers represent the subset of user readable state and writable control elements that are accessible from within the associated process' CAS. The set of writable registers is much smaller than the set or readable registers. An attempt to write a read-only register will result in the generation of a Protection exception. A small set of peer-process state is also visible via the SSR interface. As with all peer-process interaction, CAS access is gated by a process' CAS Visibility Bit Map.

Table 61: SAFE State Register Reference (SSRR) Formats

57	56	..	48	47	46	45	44	43	42	41	..	26	25	24	23	22	21	..	2	1	0
1	1	1..1	1	1	0	1	1	Target CAST							SAFE State Register Index						

Table 62: SAFE State Register Reference (SSRR) Field Descriptions

Bit Range	Field Name	Field Description
23:0	SAFE State Register Index	This field contains the Offset in the indicated CAST that is the target of the associated memory transaction. SSRs can be read and written as 1, 2, 4, and 8 byte data entities. This field is the an SSR filtered CAS operation (only a limited set of CAS words are readable and a smaller set are writable.
43:24	Target Context Address Space Tag (CAST)	When associated with a read, this field contains the reader's indication of the CAST from which it wishes to read (reading data previously placed by the indicated CAST for the reading CAST). When associated with a write, this field indicates the CAST buffer index in the writer's SM region into which the writer is placing data for subsequent reading by the indicated CAST). The associated peer CAST for the transaction is the CAST of the executing process.
57:44	Must be 0x3FFB	This field must contain the value 16_379 (0x3FFB) for the associated reference to be interpreted as an SSRR.

5.12 Module Address Spaces (MASs)

SAFE supports a forward looking Shared Code Base mechanism that employs a vector of registered, statically compiled, shareable, code modules that are dynamically linked into executables (very much like Dynamic Link Libraries). A program is composed of Modules. A module is a set of code, constants, and entry and exit vectors. At this point there are no processors that support the SAFE Module concept, however, the adoption of SAFE Modules would be a modest change to an existing architecture and adoption would significantly improve a system's ability to control execution environments, particularly their security. SAFE Modules support the ability to shift code into shared, system controlled, dynamically linked code bases that can securely perform services for a process, potentially with access to resources not available to their calling process.

A set of tables are used to translate program-specific module identifiers to system-wide Module handles. A Module Process Reference (MPR) is indexed by an 10-bit tag, termed a Module Process Address Space Tag (MPAST). A Module System Reference (MSR) is indexed by a 32-bit tag, termed a Module System Address Space Tag (MSAST). Both MPRs and MSRs ahve an associated 32-bit offset. An MPR is translated into an MSR via a program specific MPAST-to-MSAST Translation Table. A system's MSAS vector represents the set of all modules available within a system. Modules are managed by the LLS in a manner similar to that used to manage BASTs, but with a much smaller maximum code object size.

The Module Address Space (MAS) is not process specific. There are two Module types, shared, registered, System Modules; and unshared, individually created, program specific, unsharable[1], Independent Modules. The most significant bit of Shared Modules is clear and of Independent Modules is set. Modules are read-only, except for a small set of Module indices that are reserved for optional self-modification (Modules whose 16 most significant bits are all set are optionally self-modifying, making them Independent Modules as well). A self-modifying Module must also be mapped to a DAS. Changes are made through the DAS and the code is executed through the MAS. The system must be aware that self-modification is enabled and MAS invalidation occurs on DAS writes.

Modules employ Entry Points (EPs) to support calls. A Module has a vector of EPs that contain the location of a function within the Module. Externally a Module is just a vector of 4096 EPs. External calls to a Module's function are indirect through EPs. Programs that use SAFE Module EPs need not be recompiled because a constituent Module has been recompiled.

Given that the shared code base mechanism is optional, its details are not presented within this document, and are instead enumerated within a separate SAFE Shared Code Model document.

1. In this context, unsharable does not mean that a Module cannot be used by more than one program. It means that every instance of an Independent Module's use employs an independent instance of the Module and that there is no option for Module DAS sharing. It also means that an Independent Module cannot be used to provide access to system resources.

Table 63: Module System Reference (MSR) Format

<table>
<tr><td>71</td><td>70</td><td>69</td><td>68</td><td>67</td><td>66</td><td>65</td><td>64</td><td>63</td><td>62</td><td>61</td><td>. . .</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td></td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td><td>31</td><td>30</td><td>29</td><td>. . .</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="4">0</td><td rowspan="4">1</td><td rowspan="4">1</td><td rowspan="4">0</td><td rowspan="4">0</td><td rowspan="4">0</td><td rowspan="4">0</td><td rowspan="4">0</td><td colspan="19">Module System Address Space Tag (MSAST)</td><td colspan="13" rowspan="4">Offset</td></tr>
<tr><td>0</td><td colspan="12">Shared Module Index</td><td colspan="6" rowspan="3">Version Index</td></tr>
<tr><td rowspan="2">1</td><td colspan="12">Independent Module Index</td></tr>
<tr><td>1</td><td>1</td><td>1..1</td><td>1</td><td>1</td><td colspan="7">Mutable Module Index</td></tr>
</table>

Table 64: Module System Reference (MSR) Field Descriptions

Bit Range	Field Name	Field Description
31:0	Offset	This field contains the Offset within the indicated MAS of the associated reference.
37:32	MSAST: Version Index	For the indicated MAS, this field indicated which version of the MAS is being referenced. A Module can have up to 63 versions, plus a test version (Version 0). The reassignment of a Version Index requires an analysis of every program in a system to ensure that if the index to be reassigned is in use by a program, then the program is marked as Requiring Linkage, thereby preventing the inadvertent execution of a program in which obsolete linkage is employed.
62:38	MSAST: Shared Index	When MSR bit 63 is clear, this field indicates with which of a system's range of $32*1024^2$ possible Shared Modules the associated reference is associated.
62:38	MSAST: Independent Index	When MSR bit 63 is set and at least one of bits 62:54 is clear, this field indicates with which of a system's range of $32*1024^2$ possible Independent Modules the associated reference is associated.
53:38	MSAST: Mutable Index	When MSR bit 63:54 are all set, this field indicates with which of a system's range of 64*1024 possible Shared Modules the associated reference is associated.
79:64	Must be 0x60	This field must contain the value 96 (0x60) for the associated SR to be interpreted as an MSR

Table 65: Module Process Reference (MPR) Format

57	56	. . .	43	42	41	40	39	. . .	34	33	32	31	30	29	. . .	2	1	0
1	1	1..1	1	1	Module Process Address Space Tag (MPAST)							Offset						

Table 66: Module Process Reference (MPR) Field Descriptions

Bit Range	Field Name	Field Description
31:0	Offset	This field contains the Offset within the indicated MAS of the associated reference.
42:32	Module Process Address Space Tag (MPAST)	This field contains a Module specific Module Index. A Program can reference as many as 65_536 independent Modules and a Program's Module translation table contains a per Module mapping from Module relative Module indices to Program relative Module indices. An MPAST translates to a specific Version of an MSAST. There must be agreement on the specific Version of each of the Modules a Program uses.
63:42	Must be 0xFFFF	All the bits in this field must be set 65_535(0xFFFF) for the associated Process Address Space reference to be interpreted as an MPR.

6 Hierarchical Caching

The next couple pages contain a very cursory overview of caching, something that consumes dozens of pages in any reasonable Operating Systems or Computer Architecture text. Such a text should be consulted if the reader experiences anything more than an unpleasant academic deja vu and the associated textbook induced boredom (if you are learning something you should almost certainly pick up Tanenbaum's OS and/or architecture[1] book and read the relevant sections).

Caches come in five major categorizations:

1. Line Size: the number of bytes in a cache line and the granularity with which any given load/store operation is tested for cache residency (any given line in a cache can be associated with many possible data addresses and a test is required to identify a cache line's current resident). A cache's line size determines the number of low-order address bits that are not required to determine an address' cache location, for example: an aligned, 4-byte data access within a 64-byte cache line could reference 16 different locations within the cache line, but which of the 16 locations is not material to the identity of the cache line itself. The least significant 6 address bits of a 64-byte cache line are ignored for cache residency testing (those 6 bits are used to determine a datum's location within the cache line).
2. Associativity (set size): the number of possible cache locations in which any given cache line may be present. This is generally considered to be a *set size* or *way count* with:
   - a single way (direct-mapped) cache only supporting a single possible location for any given cache line (if it is not in its only supported location, then it is not in the cache). A direct-mapped cache requires that the contents of a single line be compared to the desired line to effect a residency decision, and if the line is speculatively retrieved from the cache, then the data can be made available as soon as the data's associated location in memory is determined.
   - an all way (fully associative) cache allows any given cache line to be placed in any cache location. A residency test requires that every cache line's associated address be compared to the desired address.
   - an N-way set-associative cache allows a cache line to be in N possible cache locations. For example: a 4-way set associative cache allows a cache line to be placed in 4 possible cache locations and hence a residency test requires that the 4 possible associate cache locations be interrogated.
3. Set Count: the number of address bits used to locate a cache line, or group of cache lines, within a cache. For example, a 1024-set cache with a 64-byte line size, uses address bits 15 through 6 (address[15:6]) to select the set in which the line may be resident. For an N-line direct-mapped cache, the set count is N and the set size is 1. For an N-line fully-associative cache, the set count is 1 and the set size is N. For an N-way set associative cache, the set size is N and the set count is (the number of possible line locations):

   ```
   cache_size_in_bytes / cache_line_size_in_bytes / N
   ```

   For example: a 64-kibibyte, 64-byte line size, 8-way set associative cache would have a set count of 128 (65_536 / 64 / 8)
4. Hierarchy Relationship: whether a higher level cache line is also in the closest lower level cache (inclusive), whether a higher level cache line is not in the closest lower level cache (exclusive), or whether a flexible relationship exists between a cache line's residence at independent cache hierarchy levels (hybrid).
5 Replacement Policy: how a cache decides which line to remove when a new line needs to be added. Data resides in a cache until its associated Address Space (AS) is retired or until it must be removed to make room for a competing cache line (capacity eviction). There are three major replacement strategies:
   - Least Recently Used (LRU): the cache line that has been accessed least recently is replaced. For a direct-mapped cache, every line is the most and least recently used and the LRU pol-

1. The author believes that Tanenbaum is an easier introductory read than Hennessey and Patterson or Patterson and Hennessey (two relatively more advanced texts).

icy is free. For an N-way set associative cache, there are N! (N factorial) possible usage orderings and as N increases, so does the amount of overhead associated with calculating and remembering each set's replacement state.

- Random: a random victim is selected. Research has demonstrated that without knowledge of future access patterns, LRU has the best performance, with the caveat that there are degenerative access patterns for which LRU is the worst replacement strategy. As N increases in a set associative cache, the overhead of maintaining LRU increases and the likelihood that a randomly selected cache line will be a "bad" choice decreases. When the next level cache that has only a modest performance penalty is combined with a highly associative cache (32 or more ways), the overhead of LRU may not be warranted and random replacement may have acceptable performance characteristics (this is particularly true when an axillary victim cashe (a small cache that is constructed in the belief that it associated main cache will make the occasional poor replacement decision) is present.
- Not Recently Used (NRU): the cache line to be evicted may not be the least recently used, but it is one of the less recently used lines. There are many forms of NRU and for our purposes any algorithm that is not LRU or Random is likely an NRU algorithm. For N-way set associative caches in which N is 8 or more, NRU can approach the performance of LRU, but with significantly less overhead. For example, simple NRU algorithms exit that employed a single bit of overhead per cache line plus `SetCount` counters, each `log2(SetSize)` bits in size. By way of comparison, an 8-way LRU replacement model requires management of 8! possible orderings, meaning that there are 40_320 possible LRU values per set (requires 16 LRU management bits per set). A 2-way LRU requires 1-bit per set and no lookup table and a 4-way has 4! (24 states), requiring a 24 `X` 4 lookup table (the table contains for each LRU ordering and per recently used line, the new LRU ordering) with 5 LRU management bits per set. [Note: NRU replacement models are interesting as alternatives to LRU algorithms when LRU state requirements becomes onerous. Many NRU algorithms are complex and/or time consuming and hence are only interesting as publication curiosities. If useful NRU algorithms are of interest to the reader, the *Clock* and the *FIFO with Second-Chance* algorithms are good places to start.]

Capacity eviction issues moderate the attraction of simple, low-N (an N of 2 to 4) set-associative caches (the increased probability of degenerative or competitive eviction behavior associated with low-N associativity attenuates its performance). Higher associativity caches allow the indexed address bit ranges of multiple cache lines to collide/coexist within a set, at the expense of a more complicated and power consumptive implementation. It is possible to employ a combination of NRU and Random replacement strategies that use NRU to remove the most recently used lines from consideration and then randomly replacing one of the remaining lines (for an 16-way cache, the most recently used quartet of lines can be LRU managed with replacement randomly targeting one of the remaining 12 lines).

Some levels of SAFE's hierarchical caching model employs victim caches[1] to limit the complexity of main caches (a victim cache can

1. A victim cache is a second cache at the same level as its associated main cache for which it supplies capacity eviction victim recovery. A miss in the main cache is referred to the victim cache. If the desired line is in the victim cache, then it is swapped with a line in the main cache. If it is not present in the victim cache, then a lower-level cache is consulted and when the desired line is located, it displaces a line from the main cache, which is moved in the victim cache, which in turn, displaces a line from the victim cache (victim and next level cache lookup can occur in parallel, with a victim hit short-circuiting the next level lookup). A victim cache is typically much smaller than its associated main cache and it may also employ a higher associativity. Victim caches are used when the cost of going down to the next cache level is sufficiently high, either in latency or power consumption, that local recovery from a bad replacement decision is worth the expense of implementing a secondary cache.

compensate for the less than ideal capacity evictions necessitated by simple 2 to 4-way set associative caches). The SAFE implementation detailed here prefers the advantages of simplicity and lower power consumption of 4-way caches and accepts the compensatory need for small, but relatively more complex and power consumptive smaller victim cache.

[Note: The Level-2 cache (L2) implementations in most contemporary high performance processors are L1 victim caches. Unlike a traditional victim cache, their structure is to address capacity issues in the L1, not shortfalls in the L1's eviction policy. From a SAFE perspective, an modern CPU has a 96kB L1 with a 512kB L1 victim cache, and an 8mB L2, rather than the current nomenclature of 96kB L1, 512kB L2 and 8mB L3. SAFE counts observable cache interfaces, The interface between a traditional L1 and L2 is purely abstracted by the associated processor and none of the organizational artifacts of the structure are exposed to external interactions.]

6.1 Cache Hierarchy (CH)

SAFE's Address Space (AS) centric access model is termed *logical addressing*. This is intended to differentiate it from the existing notions of virtual and physical addressing. Logical addresses exist in a process domain in which a logical address is functionally a 72-bit, globally unique address. A transparent sparse address management and protection scheme is then applied to these logical addresses within SAFE's System Components (SCs), its Cache Hierarchy (CH), and its Processing Nodes (PNs). Outside of SCs, addresses are cached and they behave exactly like 72-bit physical addresses that are accessed via content-addressable (and hence protectable) cache accesses.

SAFE's CH is a multi-layered hardware interface that efficiently migrate cache lines up and down its associated hierarchy in response to its usage. It is based on a simple directory model that tracks lines and distributes management and ownership within its hierarchy to minimize requisite inter-PN communication. The distributed cache management model is dependent on the maintenance of a strictly inclusive CH (SAFE's CH is strictly inclusive, but with relaxed line management rules that allow a modicum of carefully tracked hierarchical level skipping with a restricted associated usage model).

Each cache level employs a cache line size that is 1/4 of the next lower level cache's line size (4 times the size of the next higher level cache). For example, an L3 cache line is 1024 bytes in size, the higher level L2 cache line is 256 bytes in size and the lower level L4 is 4096 bytes in size. This structure drastically reduces the amount of cache tag overhead[1].

A canonical SAFE cache has 5 levels. The highest level (L1) cache employs 64-byte cache lines, with L2 through L5 lines size of 256, 1024, 4096, and 16_384 bytes, respectively. SAFE's per storage byte cache tag counts and sizes shrink as line sizes increase. Conversely, the growth of the cache, in terms of bytes, has exactly the opposite impact, with an 8tB L5 requiring multiple gigabytes of tag overhead (though still much less overhead than that associated with a legacy system's inode[2] and page table overheads). The respective caches employ a range of performance enhancements that are dependent on the relative speed and power consumption of the next cache level. SAFE L1 caches are expected to at least match the effective access speeds of legacy caches[3]. When all aspects of memory access are factored, all other levels of the SAFE CH are expected to be slightly faster and somewhat more power efficient than the legacy mechanisms they replace.

SAFE slows to legacy system speed when a disk access is required by an LLS. The SAFE model allows high-speed application switching without intervening movement into and out of an OS kernel. SAFE allows an application processor to transition to a new application prior to the resolution of the associated fault. An LLS does not need the complicity of the application processor to resolve an access fault. SAFE keeps relatively expensive application processors busy on applications and relegates relatively performance insensitive storage maintenance

---

1. An 72-bit system address represents a significant tag expansion in an L1 cache, relative to a 64 kibibyte, 64-byte size cache line, 2-way set associative, 48-bit physical address cache. The legacy organization requires 27 bits for its address tag, plus a couple state bits. Conversely, the same size L1 cache tag in SAFE consumes 54 bits; more than twice the legacy requirement. Conversely, a SAFE L2 cache tag has only a quarter the tag count, and even with 2x tag size, its tag overhead is only half that of a legacy architecture. The fact that the L2 is 5.3 times the size of the L1 leaves a SAFE's tag overhead only a bit more than a quarter the size of a legacy cache.
2. Inodes are part of a legacy systems storage management infrastructure. SAFE does not us inodes and has none of teh issues associated with inode availability (or lack thereof).
3. SAFE's L1 cache tag comparison is slightly slower because its wider tags, but its PR-to-SR translation is significantly faster. simpler, and lower power. Given that a cache access requires both an address translation and a tag comparison, and that the SAFE tag comparison can potentially start sooner than a legacy system's, the expectation is that SAFE's L1 caching infrastructure is no slower than that of a legacy system, with its subsequent caching levels becoming progressively faster than legacy.

and management duties to lower-cost processors that can optionally employ task specific accelerations (logic that would never be added to a general system, but that can be trivially applied at the periphery of a $5 core to give it performance parity, for very specific management tasks, with a legacy high-speed main processor, should such parity be deemed valuable).

SAFE supports cache line *sub-lining* of caches below the L1 and above the L5. A sub-lined cache line contains 4 aligned quarter-lines (for example: a 1024-byte cache line contains 4 sub-lines with 256 bytes in each). Sub-lining allows a cache line to contain 1 to 4 sub-lines of an associated cache line. A cache fault does not require the loading of an entire cache line, only the missing sub-line need be loaded. Sub-lining of large cache line sizes decreases potentially wasted data movement. Sub-lining of small cache line sizes maximizes the effective cache size, obviating the need to store unneeded portions of a cache line and allowing higher utility data to replace it.

Line-sharing allows a sub-lined cache line to contain content from two independent cache lines (concomitantly creating a 2-way set associative cache structure). Theoretically a line-sharing cache could be more than 2 associations per cache line, but it is currently unclear that there is any advantage to doing so. Line sharing allows a sub-line from either line to occupy any sub-line. A single line can occupy all of a its sub-line slots, nulling the competing line's contents. The initial SAFE model only allows a sub-line to occupy 3 of possible 4 sub-line slots.

Direct look-down uses a higher level cache to locate a line's inclusive content in its adjacent lower level cache, thereby obviating the need to perform a cache lookup in the lower level cache. This look-down requires a couple of additional cache tag bit in the higher level cache, but for those couple bits, a cache miss can require nothing more than a direct access to the targeted data within the lower level cache. This saves power and potentially decreases latency.

SAFE's cache optimizations are pair-wise mappings between higher and adjacent lower level caches. The increased descending line size represents a form of prefetch (the near data is automatically fetched with the target data). Unfortunately, look-down mapping are not transitive. A look-down can only hit[1]. A miss in a higher level cache necessitates a lower-level cache access. Each cache access can experience a sub-line hit (the requested data is returned and promoted in the cache hierarchy), a line-hit/sub-line-miss (the lower-level cache is directly indexed and the associated data is immediately returned and promoted through the cache hierarchy), or a miss (the next lower-level cache is accessed). The management of look-down state requires communication from the lower cache to the higher cache if a lower-level cache line is moved (discussed below in the victim cache and null cache sections). A direct look-down access decreases latency and power consumption when processing a sub-line miss[2].

SAFE's CH relies on a directory at each level to track and manage the residency of its associated cache lines. A CH supports the migration of a per-line Manager status within the CH, allowing an intermediate cache to act as the control point for a cache line. Manager status migrates to the highest unified cache (no peer caches share the entry) associated with a cache line. Given the manner in which cache lines shrink in size as they move towards the PNs, this Manager status can be shared across the same data by multiple caches. By way of example: an L1 can be the Manager of a 64-byte line (let us refer to it as A0) and another L1 can be the Manager of another 16-byte line within the same 256-byte L2 line (let us refer to it as A2). Both of the previously mentioned L1s can be Managers of their respective cache lines and the associated L2 can be the manager of the sub-line pair that is not resident in an L1 (Let us call them A1 and A3). A cache line's Manager can act

1. The original look-down was triggered by a hit in the line, but a miss in the sub-line. The lower level cache's sub-line holds the entire higher-level cache's line, making a miss in the lower level cache impossible.
2. A sub-line miss is differentiated from a line miss: a line miss indicates that none of the associated sub-lines for the target line are in the cache; while a sub-line implies that a peer sub-line is in the cache, but not the targeted sub-line.

autonomously with regard to the line (important for minimizing the overhead associated with the maintenance of cache coherency).

Partial Cache Line Update: Any cache line can trigger a [transitive] sub-line update (write-through). These operations can be used to clean local cache lines by pushing dirty state through to lower-level cache elements. The last (cleansing) sub-line update associated with a specific cache higher-level line will indicate a change in the state of the line from dirty to clean. When initiating a [transitive] sub-line update a higher-level cache sends a sub-line, possibly a sub-line it has received from a still higher-level cache, to a lower-level cache. Such an update includes the line size and offset. The target cache must mask the injected sub-line without disturbing the unaffected cache line bytes.

Cache Line Push and Partial Cache Line Push: SAFE CH directories makes it relatively simple to steer an updated cache line state to its current higher level directory. Rather than forcing an invalidation up a CH so that a cache line can be updated in a lower level cache, the updated state can be sent up the directory managed tree and the updates can be directly applied to the associated cache lines at their highest levels of the cache. This significantly improves performance, minimizes power consumption, and decreases wear on cache levels implemented with write sensitive storage technologies.

6.2 Victim Caches

Using expected implementation technologies, the L3 will be roughly a quarter of the speed of the L2 and roughly 16 times faster than the L4. Additionally, the L4 is "off-chip" and the performance mandated characteristics of its interface are highly power consumptive. Thus, although the relative speed of the caches, is considerable, the power implications are even more severe. Sub-optimal cache line eviction costs time and power. To this end, a Victim Cache has been incorporated into the L3, and lower level cache designs. All dirty cache lines evicted from the main L3 cache are sent to the L3 Victim Cache[1], and the oldest colliding line in the victim cache is concurrently evicted from the L3 Victim Cache to make room for it. On entry to the Victim Cache, a cache line becomes a candidate for cleaning. Spare interface cycles are used to request write-through of dirty higher level content and to write local dirty content through to the next lower level cache for the oldest members of the victim cache. This facilitates quick removal of such lines from the Victim Cache when an eviction is required. Once a Victim Cache line has been cleaned it becomes a candidate for Null Cache placement if it has active upstream cache content. The presence of a victim cache makes relatively low associativity main caches less onerous from a performance perspective, and a small but high associativity Victim Cache minimizes the impact of degenerative set-associativity behavior; for example, a 4-way cache could handle bursts of up to 20-way associativity for a small set of cache lines if their associated Victim cache is 16-way.

6.3 Null Caches

SAFE's hybrid inclusive architecture and its purely directory-based coherency mechanism creates a set of required behaviors that are not a standard part of a cache's implementation. For example, the directory structure necessitated by the coherency mechanism requires that a cache know whether any of the higher level caches it feeds contain a reference to any lines it holds. This form of inclusive caching with a directory obviates the need to perform cache probes (reading peer caches to see if a particular cache line is present), but it also increases the necessary record keeping.

Null Caches forestall the eviction of a clean, higher-level cache line because of the capacity eviction of its associated line from a lower-level cache. A fully connected coherency directory tree (inclusivity) would require the eviction of higher-level lines that correspond to a line departing a lower level cache. Very high usage, clean cache lines (code being a good example) need not be represented in all lower level caches. Such lines are likely to maintain their residence and if they are evicted, they can be discarded. This allows a lower level directory to store only the directory information without a backing cache image for lines represented by the directory. A discard in a higher level cache will result in a discard message being sent to a lower level cache, which will then update its directory. If the directory for a cache line becomes empty, then the cache sends a discard down to the next lower level cache and discards the now doubly Null cache line. Null caches experience capacity evictions just like a normal cache and if a line is evicted from a null cache, all its associated, higher-level cache lines must be discarded. However, a Null Cache's storage requirements are very small. For example, 16 Null L3 cache entries require roughly the same area that a single normal L3 cache line. If a Null Cache entry is written, then it must move back into the main cache (no different from a cache miss, except that it does not stall the associated process while it pulls content from the neighboring lower-level cache. The format of a cache directory entry is covered in the next section. Null Cache, Victim Cache, and lower level cache lookup can all occur in parallel with short circuiting of the lower level cache if either of the ancillary higher level caches hit.

1. Clean cache lines are sent to the Null Cache.

6.4 Cache Usage Models

Most modern computational systems designed for moderate-to-high performance, employ multi-level caches. In this model, a relatively small and hence relatively fast[1] Level-One (L1) cache is placed next to the CPU, with a larger and slower Level-Two (L2) being placed next to the L1. An address that misses in the L1 is sent on to the L2 and the missed line is brought up into the L1, typically forcing a line out of the L1. If an L2 lookup misses, the associated address is passed along to a Level-Three (L3) that stands next to the L2. The L3 is the end of a legacy cache hierarchy and a miss in the L3 requires a read of memory. Memory cannot "miss" because the Translation Look-aside Buffer (TLB) virtual-to-physical address translation lookup required to access the L1 cache succeeded (its physical address was known and hence the associated data must be, or more accurately, is assumed to be, at the indicated memory address).

In legacy systems, the fun stuff happens when the TLB misses (complex page-table walking hardware, that must account for the fact that the page tables are themselves paged and a table walk can repeatedly miss in memory, recursively triggering the hardware/software mechanisms that are invoked when a needed page is not in memory).

The difficult stuff happens when we add SSDs and particularly when the emerging type of Fast SSD (FSSD) enters the picture. Storage that lies between memory and hard drives does not easily fit into the legacy memory hierarchy - storage must be partitioned between SSDs and hard drives (which is grossly resource inefficient) or SSDs must be software cached (which significantly slows their access) For the emerging FSSD technologies, software management may increase access latencies by an order of magnitude. If we accept two independent realities as facts moving forward: (1) FSSDs have already become a viable technology (Intel's Optane) and will become more attractive in time (carbon nanotubes, MRAM, and FeRAM are examples of just some of the emerging technologies); and (2) hard drives (spinning rust) are not going to be replaced by SSDs (in anything but modest size systems) and for enormous storage installations, tape drives, tape libraries/vaults and/or cold storage hard drive systems will be necessary to hold the enormous amount of data being generated. The control of I/O interface data path width will strongly encourage the continued use of physical multiplexing (moving increasingly dense magnetic fields past read-write heads is the only mechanism for cost effectively realizing our storage needs in the short and medium term [it is prima facie ridiculous to make claim regarding what form this technology is going take in the long term]). That said, a SAFE LLS is associated with a system's last level of storage. An LLS is as easily integrated into a system terminated by a DRAM-based L3 as it is to a system terminated by a tape-library-based L6.

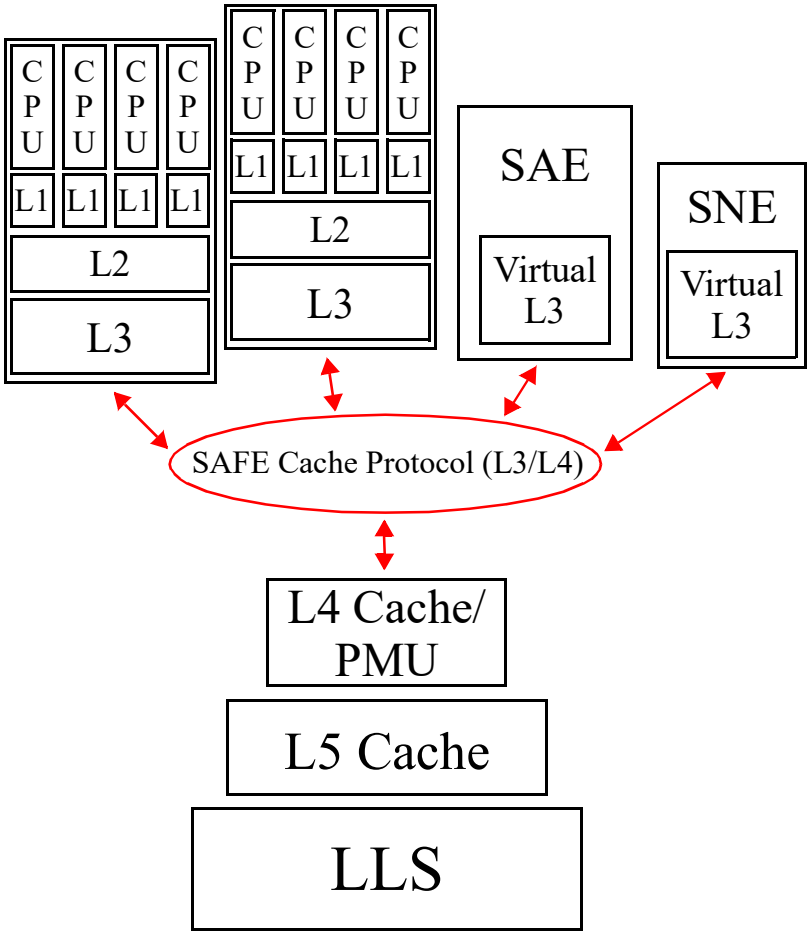
One advantage that legacy systems have over SAFE based ones, though at significant cost, is the ability to impose relatively tight resource constraints on processes. The invocation of software to control page faults allows software to monitor and control the amount of system memory a program is allowed to consume (a critical capability when dealing with a shared resource whose profligate consumption can negatively effect competing programs). The use of software within an already high overhead system-call model, minimizes the effective cost of sophisticated management. Conversely, absolute fidelity to resource constraints with a one-part-per-ten-thousand accuracy is probably overrated when 1% accuracy is typically more than sufficient. SAFE also allows its coarser control mechanisms to function at all levels of its cache hierarchy without software intervention[2].

1. As a general rule: the larger a cache the slower its latency. This is at least in part, a side-effect of additional levels of address multiplexing.

2. Using software slows the decision making process by orders of magnitude and consumes orders of magnitude more power, albeit for a very small percentage of a system's overall power consumption and processing time. SAFE changes to this model are motivated by a desire to improve security, with superior performance (speed/power-consumption) being a serendipitous side-effect.

7 System Component (SC)

A System Component (SC) is logically a hardware sub-system that supplies a core OS functionality via the system's cache hierarchy. With the exception of an LLS, most SCs attach to the Cache Hierarchy (CH) at its L4 as Level 3 caches. An SC hides its internal processing in ways that abstract a logical L3 through L1, requiring fidelity to only the L3/L4 cache interface.



The graphic to the left shows the PMU integrated into the L4 cache. While this is the right level for PMU integration, and it is a simple implementation optimization to use a common embedded processing subsystem to handle both PMU and L4 cache management duties, logically an L3/L4 cache interface exists between the PMU and L4 cache.

Secure Accelerator Units (SAEs) are a special case of SC due to their ability to support a highly dynamic configurations based on its resource partitioning. A SAE can run many independent, but small kernels, or a smaller number of larger kernels. SAFE's process scheduling infrastructure does not handle sub-scheduling of an SAE's internal resources and instead supplies a Work Queue (WQ) to the SAE and the SAE is responsible for sub-scheduling within its resource horizon. This frees SAFE from managing an undue level of GPU implementation awareness and allows SAEs to optimize their internal structure.

8 Processing Node (PN)

A Processing Node (PN) is synonymous with an application processing subsystem. Generally a PN will be a multi-core system with 3 internal cache levels (once die-stacking of DRAM becomes commonplace). A PN never runs privileged code, though SAFE's internal security mechanisms make the tasking of a PN core with less security critical tasks on behalf of an SC a no risk oprion.

Each PN has a single associated type. PNs with a reconfigurable capability either employ a hot extraction and reinsertion model to alter their type (used for PNs with high-latency reconfiguration times), or their type is fixed as a reconfigurable PN and processes within their associated Process Domain (PD) contain decorations that indicated which of set of possible reconfigurations is appropriate to (should be adopted for) the enqueued process (used for low-latency reconfiguration times).

A PN is a standard programming language (C, Java, Python, etc.) controlled, processing core that is attached to an L1 cache. In general, this is a standard, single processing core (herein termed a CPU), but on some architectures the notion of processor boundaries is blurring (hardware threading being an example), and hence SAFE nomenclature relies on a monolithic L1 attachment and associated SAFE Adaptation layer (translation tables, translation caches, PNQ interface etc.) to delineate PN boundaries. A SAFE adaptation has serial access to a vector of CASs, but only one CAS at a time. A multi-threaded CPU has one SAFE Adaptation Interface per thread. Access to a single CAS via a adaption layer implies the presence of a single core. Logically a multi-threaded core is two or more cores, one per thread.

For our purposes, no distinction is drawn between a CPU and a DSP. Neither processing speed nor processing width are material (a 64-bit 5ghz core and a 16-bit 100mhz core are identical from a PN labeling perspective, i.e. they are each functionally CPUs). If a node is generally programmable (GPUs are not generally programmable, though as indicated earlier, in an environment where everything is Turing Complete and in which the notion of "general" is fungible, "generally programmable" can be a judgment call that in the final analysis makes no real difference to the discussion). If your boat is floated by making no distinction (or blurring the distinction) between GPUs and CPUs, knock yourself out. An SAE is distinguished from a GPU by its level of autonomy. A GPU that operated independent of CPUs via a work queue and that abstracts its memory internals via an L3/L4 cache interface is an SAE. A legacy GPU fits best in a SAFE system in an integrated PN that includes a GPU/multi-GPU capability with a CPU (an APU model, whether the GPU is discrete or not).

8.1 Secure Networking Engine (SNE) Interface

SAFE employs the term Secure Networking Engine (SNE) to represent a user-level interface abstraction of a wired and/or wireless network connection. The number of such connections abstracted by an SNE is an implementation decision. SNEs represent a system physical boundary that connect via a standardized L3/L4 coherent cache layer interface that provides a standardized capability. Sockets look like sockets independent of whether they are physically supported by a quad 100 gigabit Ethernet SNE or a single 100 megabit Ethernet SNE. The option also exists to have a single physical SNE present itself as multiple SNEs (doing so does not necessarily imply any partitioning of the SNE's physical networking ports). [Note: an implementation decision to consume multiple SNE configurations also results in the SNEs consumption of multiple of the maximum of 15 L3 cache ports on an L4 cache's hierarchical cache interface.]

An SNE implementation will typically take the form of an embedded system, albeit one that, depending on the demands of the associated interface(s), may require substantially more processing capability and network stack specific acceleration circuitry than the other standard SAFE embedded systems (the LLS and PMU discussed below). An SNE may be nothing more complicated than a Raspberry Pi style multi-core ARM system with 4gB of memory and a modest speed (single, dual, or quad gigabit) networking interface. AN SNE may be a 4ghz dozen core solution with a quarter tebibyte of memory and extensive hardware accelerations that abstract a pair of 100 gigabit Ethernet interface.

SNEs are closed embedded systems that interact with the associated system at an L4 cache interface, but that are fundamentally independent therefrom. The same SNE could, without modification, be integrated into a broad range of systems. This is one of the significant strengths of the SNE model, an SNE is a generalized abstraction of networking that is independent of the associated system's host operating system.

SNEs do not run user code[1] and are therefore fundamentally different from CPUs and GPUs. SNEs rely on SAFE's MN capability for network connection set-up and tear-down requests and for metadata updates (notifications regarding changes in the state of existing SNE Sockets). Process to SNE communication is predicated on the process being granted access to the SNE's CAST (without direct SNE access a process must route connection requests through an OS module that mediates network connections.

All SAFE cache transactions have fields that identify their source and target node indices (this is part of SAFE's physical protocol mechanism that is not visible at the process/application/user level). An SNE is/has a physical L3 cache node index for physical cache transactions. An SNE is assigned an exclusive/captive vector of NSASTs (for user level network connection's logical reference cache transactions), and a CAST (for side-band communication via SM and MN transactions between a process and the SNE as a connection management capability). An SNE's physical index routes cache traffic and the logical index allows the SNE to operate within the system as a software entity and hence as an optionally visible (via configuration) peer to any other software process.

The connections with which SNEs typically inter-operate have their own NPASTs that are translated by a PN into a 28-bit system wide NSAST (8 bits identify the SNE and 20 bits identify the connection within the indicated SNE). An NSAST range is allocated to an SNEs as an aligned 1024^2 connection block. This mechanism allows a simple policing of SNEs traffic (both as a source and a target). If one end of a NSAST is connected to a PN, that PN is mapped to the NSAST via a

1. Software Defined Networks (SDN) tends to be more the purview of switches than endpoints. SAFE does not reject the notion of endpoint SDN support, but it does argue that SDN configuration tends to be an intra-SNE, externally configured mechanism, rather than code based, and it is generally fairly persistent. Hence, an SDN endpoint capability could/should be vetted for security and become a system level, not a user level capability.

NPAST's SNE Index. SNEs are only allowed to communicate within their NSAST range (SNEs use an alternative model for communication between each other).

Like other physical PNs, and as previously indicated, an SNE communicates with processes via user level ASTs (in this case an NPAST that is translated to an NSASTs), but also has a user level connection independent SNE CAST that can be targeted by the MN and SM interfaces to effect Side-Band Communication (SBC). SBC allows a process to independently (relative to legacy OS mediation) request the creation of a network connection and to later request its termination. This creates a model in which a process can request a socket, use the socket, and then tear-down the socket; all via direct communication with an SNE.

8.2 Secure Acceleration Engine (SAE)

Within a SAFE system, a Secure Accelerator Engine (SAE) an independent processing system that is differentiated from a CPU/PN by the structure of its operation (it is not a standard processing system). An FPGA-based sub-system that is possible of independent operation and that adheres to an L3 cache interface is an SAE. A more common form of SAE is a combination of a function limited and expanded GPU with its I/O interfaces (e.g. USB 3.X and/or thunderbolt) optionally expressed as independent network interfaces (see SNE above). An SAE graphical capabilities are its acceleration capability expressed as a Remote Procedure Call (RPC) interface. A graphics library expressed as an RPC is an excellent example of an SAE. Conversely, an OpenGL interface on which users execute their own code is a PN. As is a GP-GPU style node that supports OpenCL and/or CUDA. An SAE acts as an abstraction and security boundary within a system and its clients employs RPC, pre-arraigned DASs, and SM to communicate with it and it uses MN pre-arraigned DASs, and SM to communicate back to its clients.

Like a CPU, a SAE is defined by the same monolithic L3/L4 cache boundary connection to SAFE's CH. Any internal caching is private to the SAE and the SAE need only conform to the CH model as an L3 cache (this is the same as a PN or SNE and this document's exposure of a PN's L3-L1 caches is for illustrative purposes). As with the SNEs, an SAE has an index (SAE Index) that is used to route data within the L3/L4 cache interface. An SAE has a vector of clients and all SAE communication is integrity checked (coming from a properly identified source and headed to a properly identified target).

As is the case for a PN, an SAE, as a monolithic entity (a single chip can contain many such entities) that is defined by its CAS interface. SAEs are autonomous. A PMU, for example, can request service, but it has no SAE internal scheduling authority. Any level of preemption supported by an SAE is implementation specific. Concurrent scheduling between PN(s) and SAE service execution is not materially different from concurrent CPU scheduling between CPUs.

8.3 Accelerator

From an historic, CPU-centric system's view, SNEs and SAEs are accelerators. This view holds that there are only two types of processors: CPUs and accelerators. Conversely, given Minsky's minimal hardware requirement for Turing Completeness, my electric toothbrush may qualify, and the pedestal on which CPUs have placed themselves is probably unwarranted. Processing is commoditized. The integration of standard processing elements with special purpose acceleration logic is how modern Application Specific Integrated Circuits (ASICs) are built. The commodity FPGA vendors integrate standard processor s into some of their FPGA chips, and support soft core in all their chips. Moving forward, it probably makes more sense to discuss functionality than circuit topology, almost all modern chips are hybrids of general/standard logic that transcends the associated chip's function/purpose, and logic that is purpose built to the specific demands of the associated chip's purpose/function.

Hence, we partition processing into: highly-flexible, general-purpose processing nodes (CPUs); acceleration processing with some general-purpose processing capability (SNEs); network processing (SNEs); and functional processing that is neither highly network nor acceleration centric (Accelerators). A reconfigurable Accelerator that dynamically adopts an SNE or GPU functionality is not considered to be an accelerator while acting as a SAE or SNE, but it will dynamically become an accelerator if repurposed. Hot-plug events are associated with a node's reconfiguration. A node dynamically adopts the persona of its configured device type. Although the distinction is not significant, per se, we only adopt this terminology to in an attempt to forestall "what if an X is functioning as a Y?" questions.

Standard processors have ever increasing amounts of acceleration logic attached. The *SAFE Accelerator Test* concerns whether a node: has its own independent software control model, can autonomously move data within the associated Cache Hierarchy (CH), and can operate as a target for task scheduling. If all 3 criteria are met, it is considered to be an SAE. If a device's behavior is tightly tied to the operation of another processing entity such that it can neither move data independently nor execute independently, then it is considered to be a subsystem within its associated device's execution model (its operation is subsumed by, and is inextricably connected/controlled by, the operation of the associated processing capability). It is understood that the distinction is not necessarily as binary as it may seem. At issue is whether the associated device has its own process scheduling mechanism, or whether it is tied to another device's scheduling.

8.4 Last Level Store (LLS)

At a very high level, a Last-Level Store (LLS) handles half of the OS functionality subsumed by SAFE (memory management and file management, though it lacks support for a hierarchal file system). The 50% mark is a very coarse functional metric, in terms of complexity and associated work, the LLS is barely a quarter of SAFE's operational abstraction (a PMU is only somewhat less complex than an LLS, while an SNE is significantly more complex than either).

An LLS manages a system's persistent'ish[1] store. While it is possible for a system to contain multiple, independent LLSs, the norm will probably be a single LLS. An LLS represents the lowest-level termination of SAFE's Cache Hierarchy (CH). An LLS is not a cache[2]. It is the backing store that makes caching possible.

It is possible that an LLS could be a purely hardware construct, but highly unlikely (too much complexity/effort for too little benefit). A reasonable approximation of an LLS is a Raspberry PI 4 like subsystem containing 4 processing cores running at 1.5 to 2 ghz, 4 gibibytes of DRAM, enough I/O to support the attachment of a dozen or more hard drives, a multi-gigabit Ethernet link, and one or two connections to a SAFE caching hierarchy. In volume, an LLS would cost less than $50 using a 2018 SoC costing models.

Initially an LLS will be a pure software construct running on commodity processing cores. In time, a modest amount of FPGA based acceleration will be added to the system. The more successful (performance relevant) acceleration implementations will migrate to ASIC/SoC implementations over time. The addition of acceleration hardware will lower overall LLS cost (probably somewhat higher capital expenditure that is more than covered by a compensating decrease in operational expenditure, largely power consumption). On small systems, a single core LLS that is integrated into the associated system's flash drive will be capable of handling its system responsibilities.

An LLS is an embedded system whose standardized external interface can be supported by a potentially broad range of independent solutions. While an LLS is a trusted SAFE component, it is not omnipotent within a system. LLSs have a range of responsibilities. PMUs and SNEs have independent ranges of responsibilities. An LLS must respect SAFE's domain partitioning and there is nothing an LLS can do to force a PMU, SAE, or SNE to comply with its wishes. A form of checks and balances exist between SCs. Each piece of a SAFE system's distributed OS is independent of every other piece.

A BAST is inextricably tied to an LLS. On a fundamental level, an LLS is a BAST management system. A single LLS is somewhat arbitrarily limited to a single 64-bit address space (16 exbibytes) and/or a billion data objects (the limit of a 30-bit BAST), whichever limit is reached first, with multiple LLSs required to bring a single system's storage capability beyond these limits. An LLS could manage a larger data store, but as the amount of data fronted by an LLS grows, the LLS itself increasingly becomes a choke point for access to its data.

SAFE supports the integration of multiple LLSs within a single system. By employing a vector of LLSs (up to 64), a SAFE system can manage an indefinite number of data objects, but independent of the number of LLSs available, a single SAFE system is restricted to a billion active data objects. An LLS can be partitioned into a set of peer LLS nodes, each of which manage a specific range of DSASTs within a co-operatively managed BAST vector. Such a solution has multiple L4/L5connection options to the associated partitioned L5s and potentially shared L4s.

1. Data on any system has a highly variable persistence profile. Some is clearly ephemeral, while other maintains an extreme level of constancy. In this instance, persistant'ish is best characterized as an optional trait associated with the ability to create persistence, whether that option is employed or not.
2. An LLS may contain caches, but if it does, they are transparent to the associated system. Caching within an LLS is an LLS performance optimization that improves LLS characteristics, but is not a requirement for the LLS itself.

Multiple SAFE systems can function as multipliers of this concurrent data modification limit. Network connections between independent SAFE systems will support the dynamic mapping of data objects on remote systems into a local system's GAST horizon, but without coherency between the individual SAFE systems (data migration is easy - naive, system spanning, coherent, distributed data processing on a individual data object basis is hard). Conversely, a system's data space can be highly dynamic, moving data objects between systems on an as-needed basis.

A BAST is a not necessarily linear vector. A BAST's block hierarchy can have nulls (one or more invalid reference within a data object). Referencing a null results in a Protection Exception. Nulls are explicitly created by releasing blocks within a contiguous allocation (the option of crating nulls is not the normal case and it exists as a remediation of the general case, not as an integrated capability). A data object can have as many nulls as they have backing pages.

The LLS External User-Level Interface supports calls like:

- GAST Create: creates a new, unique random GAST value

```
boolean gast_new (
    id*     gast, // pointer to gast index
    config  c,    // configuration
);
```

- GAST Duplicate: creates a new, unique random GAST value and points it at the input GAST, but with the indicated access characteristics. Access characteristics must be compatible with the set of access modes indicated by the AST's owner.

```
id* gast_dup (
    id*     gast, // pointer to a GAST to be copied
    config  c,    // configuration
);
```

- Open and Close are like the legacy open and close, except they take a GAST and return a pointer.
- Read and Write are like legacy read and write, using the DPAST based pointer value returned from the open call.
- GAST destroy: delete a GAST, and if it is the only remaining reference to the associated BAST Table entry, remove the referenced data object from the BAST Table as well

```
int gast_free (
    gast*  x
);
```

Data objects are not named, beyond their GAST indices, within the SAFE infrastructure. Data object naming is an artifact of a user-level file system like interface[1] and not a part of a hardware based access and security model. Any given data object can have numerous different alpha-numeric names and exist in an arbitrary number of otherwise independent environments (user spaces). Naming, as opposed to indexing) is an aspect of a User Interface, the only major OS component not totally or largely subsumed by SAFE.

1. As will be detailed later, SAFE does not support the legacy notion of a file system. SAFE supports the creation of an indeterminate number of mappings of data objects to Virtual File Systems, which are representations of an access capability to a subset of a system's GASs.

8.5 Process Management Unit (PMU)

A Process Management Unit (PMU) manages a system's process execution. It does not create/instantiate processes, but once a process is created it is handed to a PMU for execution management. A PMU automatically, from the associated system's perspective, decides: in what order the set of existing processes will be executed; how long a process will be allowed to execute without interruption; and, if a process becomes blocked, it manages its re-introduction into the set of executable processes. A system can have more then one PMU, but only a very large system ever would. A PMU is integrated into a SAFE system's L4 cache and represents a control/management node that is independent of the system's LLS, but one that is similarly responsible for performing legacy OS duties.

Like an LLS, it is possible, but it is highly unlikely, that a PMU could be purely a hardware construct. Like an LLS, a reasonable approximation of a PMU is a Raspberry PI 4. Unlike an LLS, a PMU connects into a SAFE system much higher in the CH. A PMU's integration with the L4 is both upward facing into the resident vector of L3 caches, and downward facing into the L5. In a multi-PMU system, inter-PMU communication occurs via a common ring that connects the PMUs to the L5 the lower levels of the CH. PMUs are NCHCs and as such have an associated CAST that allows a PMU to interact with process and other NCHCs and with system's processes. In volume, a PMU would cost less than $50 using a 2018 SoC costing models.

Initially a PMU will be a pure software construct running on a vector of commodity processing cores. In time, a modest amount of FPGA based acceleration will be added to the system. The more successful acceleration implementations will migrate to ASIC implementations over time. The addition of acceleration hardware will lower overall PMU cost (probably somewhat higher capital expenditure that is more than covered by a compensating decrease in operational expenditure (power consumption). On small systems, a single core processor that shares LLS and PMU duties, and that is integrated into the associated system's flash drive will be capable of handling its system responsibilities.

Relative to a legacy OS, a PMU subsumes a system's process management infrastructure. A PMU handles all active scheduling decisions, including "cron" style scheduling. Although the PMU supplies only a single of the containerized OS's functionalities, its process control mechanism represents a significant additional level of security that the LLS cannot. The PMU also supports SAFE's encryption model, in which data present in persistent storage technologies contain encrypted data and that data is not decrypted before it is stored into its L3 caches.

9 Scheduling

SAFE manages a vector of processes within a set of Work Queues (WQs) and using a Blocked Queue (BQ) and a Time Queue (TQ) to feed the WQs. Processes are either runnable and present in a WQ, waiting for an event that indicates the availability of a needed resources on the BQ, or waiting for a wall-clock time to transpire on a TQ. There are many implementation details that are not part of the SAFE specification, but it is required that some form of Processing Domain (PD) specific set of WQs be implemented with a BQ and TQ capability. How these are achieved is implementation specific.

9.1 Processing Domains (PDs)

SAFE supports a set of independent Processing Domains (PDs). A PD represents an executable binary compatibility shared by all its member PNs. PDs represent a mapping of processes (software) to execution capable PNs (hardware). Each process has an associated PD. Each PN can support one or more PDs. A multi-PD capable PN adopts a process' PD (personality) while that process is executing, potentially shifting to match the requirements of a different PD before executing its next process.

Multiple PNs can share a PD if they can execute against the same code base. An ARM and an x86 processor cannot both be associated with the same PD, unless a PD involves an abstraction above that of computational architecture, for example a Java Virtual Machine. A CPU and GPU are in different PDs. A 32-bit x86 architecture PD can include a 32-bit x86 architecture PN and a 64-bit x86 architecture PN. A 64-bit x86 architecture PD cannot include a 32-bit x86 architecture PN with its naturally associated a 64-bit x86 architecture PNs.

A multi-PD PN can potentially support a moderate number of PDs: the aforementioned ability of a 64-bit processor to support compatibility with 32-bit code (true of many different processor architectures), the ability to execute in expanded instruction set modes (MMX, SSEx, AVXx, etc.), and the ability to execute in restricted execution modes (e.g. ARM's Thumb). These each represent a potentially unique PD. Given the plethora of architectural extensions, the set of PDs could become extensive, though in practice it is relatively limited. PD diversity is typically a statement about a system's PN heterogeneity. The cost/complexity of maintaining a diverse set of PNs typically transcends the value of doing so.

Each PN's execution behavior is controlled by its associated PMU. A PN executes until its PMU tells it to stop, wither because it has become blocked[1] or it has exceeded its quantum. When removing a process from a PN, it provides an alternative process or it idles or halts the PN. In all cases, the PN is a passive agent with minimal scheduling overhead or capability, aside from its ability to swap processes.

As a PN design becomes more aligned with automatic process management, PNs will begin to support background context swapping. In this model a PMU will instruct a PN to stop while the PMU switches out the PN's current process to its CAS and then associates a new CAS with the PN, loads its state into the PN the triggers the PN to execute. A multi-threaded hardware core will execute an alternate thread while the PMU swaps processes. This dedicates relatively less expensive PMU core(s) to overhead activities and allows relatively more expensive PN cores to run programs. With small amounts of acceleration hardware and with heavy reliance on a caching of CAS state, relatively few PMU cycles will be required to swap a process out of a PN and to swap another process into it. With simple DMA support, a single core PMU will be able to keep dozens of processing cores active. Ultimately, a PN will have no relative advantage over a PMU when context switching and it will be more power efficient to passivate a CPU than to use it to move process state.

1. A PMU sees that a process has becomes blocked when it sees a request at its associated L4. A miss in an L4 will typically trigger the transition of a PN's process out of the PN and into a blocked state, pending the outcome of the L4 cache miss.

Managed code[1] is not a PD issue, per se. An OS instantiates a specific PD bound instance of an interpreter or Virtual Machine (VM) when a decision is made to execute a managed code program instance (Python, Java, Ruby, etc.). Once instantiated, a process cannot transcend its PD. Conversely, a multi-threaded program or multi-processing software system, does not necessarily need not execute all its operational components within the same PD.

Fat binary[2] execution is either an instantiation decision or a run-time decision, based on the supported execution models. New process creation falls outside the purview of SAFE and hence any dynamic actions will be accounted for by the PD of a WQ within which a process is placed. Scheduling rules must maintain PD associations (a process started in one WQ must never be scheduled to run in a WQ with a conflicting DP).

9.2 Context Switching

Twenty years ago, fine-grained scheduling made sense. In general, there was only one processing core and it was relatively expensive. In modern systems (circa 2020), when a dozen cores are commonplace and servers may have 50 or more, a modern scheduling model needs to account for this new reality. Context switching is not free. On legacy systems, the cost of storing one process' CPU state and loading another process' can be dwarfed by TLB and cache miss effects associated with loading a process' working set.

All the delays associated with a process recreating its working set in the local caches must be included in context switching overhead. Saving the previous process' register state and loading the next process' register state may consume no more than a couple microseconds (less on architectures with shadow registers that facilitate the introduction of short lived contexts). However, the larger a process' working set, the larger its associated context switching overhead. A couple microseconds of register movement can easily run into hundreds of microseconds of cache delays necessitated by the reconstruction of a process' working set. Historically, context switching costs have tended focus on latency, but power consumption is becoming equally important. The considerable data movement associated with the movement of data from and to caches, particularly given that this data was generally already cache resident when the process was most recently swapped out, consumes considerable amounts of otherwise wasted power.

9.3 SAFE Process Management

SAFE employs a legacy, 3-process state model[3]: running, ready, and blocked. SAFE does not support software threading, per se. On a certain level, all SAFE processes are threads, or capable of being threads, based on SAFE's low cost memory sharing model. SAFE relies on its simple table based, hardware managed process state (CAS) to replace complex, high overhead, software-based process management. SAFE's hardware managed processes are no less capable than their legacy counterparts, but they so have a lower overhead cost and they are less subject/prone to error.

1. Microsoft coined the term *Managed Code*, and within that original context, it has a specific meaning. In the broader context, it has become synonymous with code based on languages that are interpreted or that are compiled into and intermediate (software defined) virtual machine (some form of byte-code). This execution model is associated with programming languages like Java, Python, JavaScript, Lisp, Perl, and Ruby. The source code for such a program is bound to a PD dependent interpreter or virtual machine when it is instantiated.
2. A *Fat Binary* contains multiple, processing domain specific, multi-threading execution options. For example, a ray tracing model that can use GPU specific ray tracing kernels and employs the solution that matches the capabilities of the available accelerators, or that uses a software library in the general-purpose execution environment if acceleration is not an option.
3. The more recent and sophisticated 5 state model is not employed because SAFE does is not suspend its processes to disk.

On a legacy system, a process shares (inherits) the page table of an associated process that falls under a common threaded process umbrella. Between threads of a common process, a lower-cost context switching model and offers faster and lower power inter-thread communications (using shared memory and thread aware socket models). In these legacy software threading environment, the respective call stacks for every thread within a process are allocated within (from) a common memory block (generally a bad idea, even with carefully managed thread dependencies that serialize the execution of threads, and the associated data hazards). Any thread can, without any oversight or external visibility, interact with the call stack of any other thread within the same process. SAFE supports this model, but is also makes it possible for a trivial configuration change to give each thread its own private call stack, while preserving a shared/common heap model.

Within a SAFE system, a process and a thread are synonymous, though we tend to focus on the term process. Threading[1] is a matter of AST sharing that has no impact on a run-time system. The difference between a processes and a thread is a matter of configuration that has no ancillary run-time artifacts, though the inheritance rules for spawning a new thread are different from the AST inheritance rules for spawning a new process (this is still and AST configuration issue and the distinction is for easy of inheritance specification - a spawned thread inherits the bulk of its spawning process' ASTs by default and a spawned process does not inherit the bulk of its spawning processes' ASTs by default).

1. Threading is a term with two radically different meanings within the same general technical lexicon. Hardware threading represents a processing core that is capable of concurrently maintaining the processor state of 2 or more execution contexts and rapidly switching between them in response to low-level hardware delays. SAFE is hardware threading agnostic. Hardware threading behaves as if multiple physical cores are present, as it does in legacy architectures. Software threading represents the notion of multiple processes that share a principal memory allocation and therefore have a lower context switching overhead. Within SAFE's operational model, a process and a software thread are the same thing. The memory context sharing that characterizes threading is managed within SAFE by ASTs. A software thread, as such, has no run-time artifacts and the distinction between a process and thread is meaningless.

9.4 SAFE Cache Hierarchy Consistency

We use the term consistency to refer to the coherency of a cache hierarchy over time (as opposed to coherency which is a statement about instantaneous cache content). There are two principal issues associated with cache consistency:

- Power Loss: SAFE systems employ distributed processing within their memory hierarchy and are not dependent on their CPUs alone to manage system state in the event of power loss. Modern computers tend to have access to reliable short-term power delivery via integrated batteries or via uninterpretable power supplies. A sophisticated and somewhat expensive to implement active system logging mechanism could be applied to SAFE, but doing so would be an complex solution to a simple problem and one that is largely solved by existing system power architecture and standard usage models. SAFE systems require a short term battery or (ultra-)capacitor based power management system that sufficient energy to allow the system to invalidate all non-persistent system state (the L4 and L5 are persistent and hence only dirty data needs to be pulled from higher level caches into the L4 and/or L5 and/or LLS; and the LLS needs to save any dynamic state it contains. The L4 becomes the critical resource in this instance because of its speed and the L4 can jettison all clean cache state and start to push dirty cache state down into the L5 while pulling dirty cache state from its higher level caches. A SAFE system can be restarted from a consistent, persistent state.
- Component Failure: On a fundamental level, a computational system cannot have a generalized, hardware-based solution to component failure, short of system replication. A system's response to subsystem/component failure is highly component centric. Some system failures are relatively easily handled (a broken Ethernet cable can be overcome by sending traffic across a working alternative cable, if such a cable exists). A broken CPU, if it is the only CPU in a system is a catastrophic failure. Software cannot respond because the software execution platform no longer functions. Depending on the nature of the failure, it may be possible for the system to respond to such a failure as f it is a power failure and to create a consistent data state by pulling data from the CPU's caches, but the nature of the CPU failure may preclude access to its caches, in which case a certain number of ASTs are potentially corrupted, and depending in the system's management model, unrecoverable. In general terms, component failure falls into the realm of software-based failure recovery that is dependent on a failure aware data management model (the previous versions of files are preserved until a later version can be fully committed so that an application can be rerun if the application fails before it completes - basically the same model used to manage software failure).

SAFE's Cache Hierarchy Consistency Mechanism requires the delivery of an active-high Power Good signal to all system nodes. If the Power Good signal drops to ground, the node will immediately initiate its reset protocol, with the LLS instigating a claw-back of all DAS content. NASTs and CAST are inherently transient and they are sacrificed to maximize the resources available to DAST preservation.

Appendix A: Base/Bias Fields

Table 67: Base/Bias Format

X+Y-1	X+Y-2	..	Y+1	Y	Y-1	Y-2	..	1	0
Base					Bias				

Table 68: Base/Bias (6,4) Field Descriptions

Bit Count	Field Name	Field Description
Y	Bias	The generated Base/Bias value is: `(~Bias) ? ((1 << X) | Base) << Bias : Base`
X	Base	

SAFE makes extensive use of what is termed Base/Bias fields. This is a model in which a generated field value is used in which the value's resolution decreases as its size increases. It is a form of a parts-per-X resolution model, where the size of the Base determines the resolution (parts-per-X) and the Bias determines the range. There are three models for Min-Base/Max Bias: `0`, `1 << X`, or an instance specific special value, like invalid.

Appendix B: Value/(Base/Bias) Fields

Table 69: Base/Bias (6,4) Format

X+Y-2	X+Y-3	X+Y-4	..	Y+1	Y	Y-1	Y-2	..	1	0
0	Value									
1	Base					Bias				

Table 70: Base/Bias (6,4) Field Descriptions

Bit Count	Field Name	Field Description
X+Y	Value	When the most significant bit of teh ValueThe value of the field is this field
Y	Bias	The generated Base/Bias value is: `(~Bias) ? ((1 << X) | Base) << Bias : Base`
X	Base	
1		

SAFE makes extensive use of what is termed Value/(Base/Bias) fields. This is a model in which a large value range is supplemented by a Base/Bias field in which a generated value's resolution decreases as its size increases. It is a form of a parts-per-X resolution model, where the size of the Base determines the resolution and the Bias determines the range. There are three models for interpreting a Value==0 state: `0`, `1 << X`, or an instance specific special value, like invalid.

Appendix C: CAS Memory Structure

Table 71: Access Structure Format

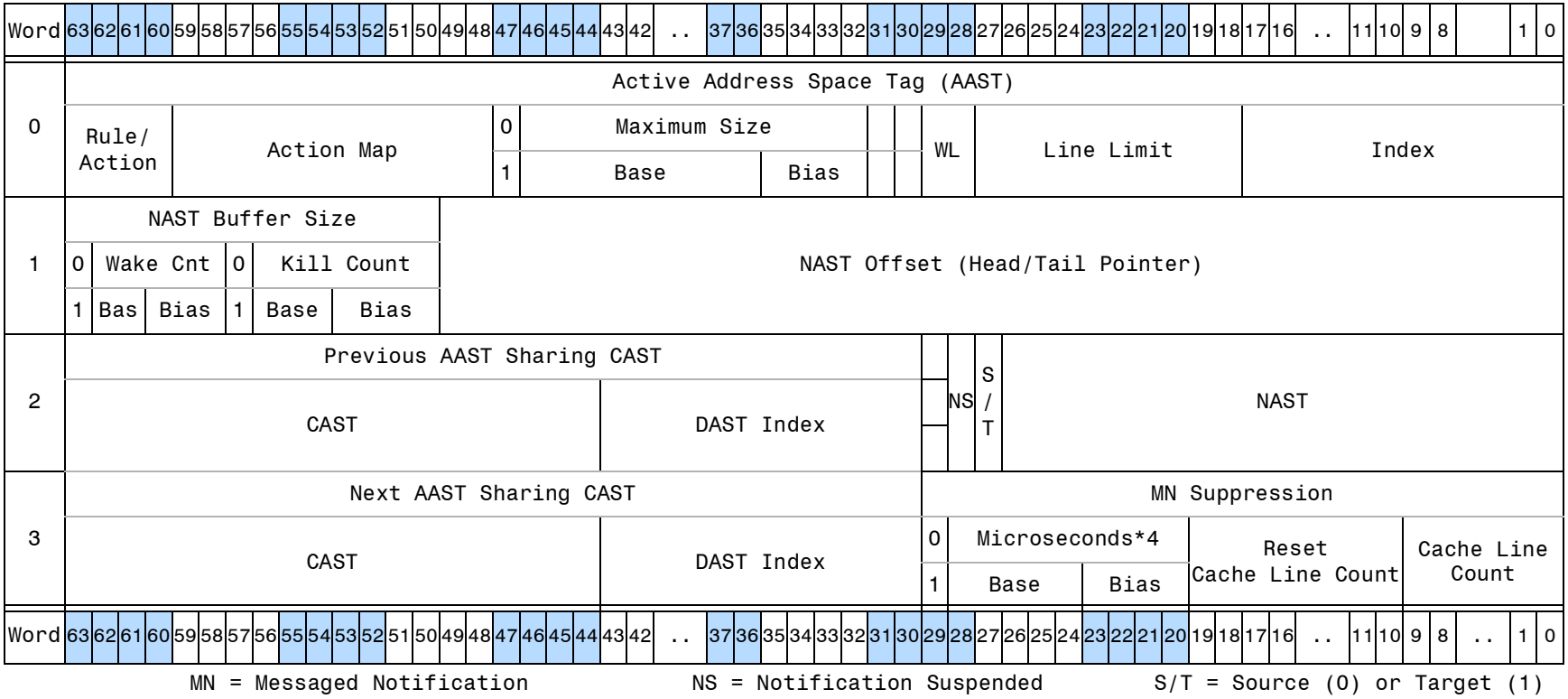


Table 72: Process Management Format

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	...	1	0

Table 72: Process Management Format

	B		Default Priority	Current Priority	Default Privilege	Current Privilege	

B = Blocked **SSEC** = Scheduling Entry Count

Table 73: Messaged Notification Queue: Data Word 0 Field Descriptions

Bit Range	Field Name	Field Description
47:0	Scheduled Time	This field indicates the most recent time (for a process with multiple concurrent scheduled times, the latest one in time) at which the associated process was scheduled for execution (the Scheduled flag is set), or the time at which the associated process is to be scheduled for execution (the Scheduled flag is clear). Scheduling resolution employs a 10ns quantum. The scheduler wraps every 8+ days (a process cannot be scheduled more than 8 days in the future (a scheduling process can be used to store and then longer term processes).
50:48	Current Privilege	This field contains the privilege currently assigned to the associated process. This field can be changed by a process' execution, typically temporarily and in response to the system moderated escalation or de-escalation of execution privilege, as required by the code module being executed.
53:51	Default Privilege	This field contains the privilege assigned to the associated process. This field cannot be changed by a process' execution, though the controlling PMU can alter it.
56:54	Current Priority	Priority concerns the ability for a process to gain access to a core and its ability to maintain access to a core. Current Priority is state dependent: if a process is currently blocked, (Blocked flag is set), this field contains the scheduling priority associated with the process; if a process is currently runnable or running, (Blocked flag is clear), this field contains the scheduling priority associated with the process
59:57	Default Priority	This field contains the associated process' default priority. This field is not altered by a process' execution. Although the controlling PMU can alter it, it does not as a general rule do so (except in response to a fundamental change in a process' role in a system's execution profile).
60	RESERVED	
61	Blocked (B)	This flag, when clear, indicates that the associated process has been placed in a ready or running state. When set, this flag indicates that the process is currently blocked.

Table 73: Messaged Notification Queue: Data Word 0 Field Descriptions

Bit Range	Field Name	Field Description
63:62	Scheduling Entry Count	This field contains the number of scheduling entries associated with the process. Every process has at least one, except for a process that is no longer active (or that has been deactivated). When this field's value is greater than 1, a triggered scheduling event, decrements this counter and discards the associated scheduling event (the associated process' state is unaffected by the event).

At the indicated index the Forward CAST identifies the CAST of the process to which the

Minimum Maximum Buffer Size is 8192 bytes

suppression resolution minimum 2 lines, 4us;

maximum 511<<7 lines, 4.45 min

NAST kill: minimum: 32@1kB; maximum 16m@1k; 0 = disable

NAST warn: minimum 1@1k; maximum 128k@1k

Every process and Interrupt Capable Hardware Component (NCHC)[1] has an associated CAS and CAST. NCHCs are assigned fixed CAST values and associated CAS structures at boot time. A process is assigned a CAS/CAST during process creation. A CAS is managed by a system's PMU(s). CASs are relatively large (they employ a 2 mibibyte structure), but not particularly expensive to manage. For most processes, a CAS is a sparse address space with uninitialized regions controlled by invalid cache accesses.

Profile based (2 bits per category:

- NAST Count,
  - 0:
  - 1:
  - 2:
  - 3: 64k
- DAST Count,
  - 0:
  - 1:
  - 2:
  - 3: 14k
- CAST Count,
  - 0:
  - 1:
  - 2:
  - 3: 2m
- MAST Count
  - 0:
  - 1:
  - 2:
  - 3: 64k
- Register Count
  - 0:
  - 1:
  - 2:
  - 3: 2m
- Private Local Store Size

1. The ability to handle an interrupt is assumed to imply an independent processing capability and hence a required standard configuration mechanism.

- 0:
- 1:
- 2:
- 3: 2m
-
-
-
-
-

Program Linkage has 3 dimensions:

- (X) Total number of referenced Module
- (Y) Total number of Modules with External Module References
- (Z) Maximum number of Modules referenced by any single Module

```
x = log2(X)
y = log2(Y)
z = log2(Z)

ref get_mod_ref (mod, ref):
   if X <= 256:
      return prog_mod_tab.u1[(mod << z) + ref]
   else:
      return prog_mod_tab.u2[(mod << z) + ref]


prog_mod_offset = 1 << y << z << (x > 256) >> 2


ref get_sys_ref (mod):
   return prog_mod_tab.u4[prog_mod_offset + mod]
```

Table 74: Program Linkage Table (Maximum Configuration: 65_536 Modules / 2560kB)

	Byte															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00_000-	Module 0 ModRef[7]		Module 0 ModRef[6]		Module 0 ModRef[5]		Module 0 ModRef[4]		Module 0 ModRef[3]		Module 0 ModRef[2]		Module 0 ModRef[1]		Module 0 ModRef[0]	
0x00_001-	Module 0 ModRef[15]		Module 0 ModRef[14]		Module 0 ModRef[13]		Module 0 ModRef[12]		Module 0 ModRef[11]		Module 0 ModRef[10]		Module 0 ModRef[9]		Module 0 ModRef[8]	
0x00_002-	Module 0 ModRef[23]		Module 0 ModRef[22]		Module 0 ModRef[21]		Module 0 ModRef[20]		Module 0 ModRef[19]		Module 0 ModRef[18]		Module 0 ModRef[17]		Module 0 ModRef[16]	
⋮	⋮															
0x00_07E-	Module 0 ModRef[1015]		Module 0 ModRef[1014]		Module 0 ModRef[1013]		Module 0 ModRef[1012]		Module 0 ModRef[1011]		Module 0 ModRef[1010]		Module 0 ModRef[1009]		Module 0 ModRef[1008]	
0x00_07F-	Module 0 ModRef[1023]		Module 0 ModRef[1022]		Module 0 ModRef[1021]		Module 0 ModRef[1020]		Module 0 ModRef[1019]		Module 0 ModRef[1018]		Module 0 ModRef[1017]		Module 0 ModRef[1016]	
0x00_080-	Module 1 ModRef[7]		Module 1 ModRef[6]		Module 1 ModRef[5]		Module 1 ModRef[4]		Module 1 ModRef[3]		Module 1 ModRef[2]		Module 1 ModRef[1]		Module 1 ModRef[0]	
0x00_081-	Module 1 ModRef[15]		Module 1 ModRef[14]		Module 1 ModRef[13]		Module 1 ModRef[12]		Module 1 ModRef[11]		Module 1 ModRef[10]		Module 1 ModRef[9]		Module 1 ModRef[8]	
0x00_082-	Module 1 ModRef[23]		Module 1 ModRef[22]		Module 1 ModRef[21]		Module 1 ModRef[20]		Module 1 ModRef[19]		Module 1 ModRef[18]		Module 1 ModRef[17]		Module 1 ModRef[16]	
⋮	⋮															
0x00_0FE-	Module 1 ModRef[1015]		Module 1 ModRef[1014]		Module 1 ModRef[1013]		Module 1 ModRef[1012]		Module 1 ModRef[1011]		Module 1 ModRef[1010]		Module 1 ModRef[1009]		Module 1 ModRef[1008]	
0x00_0FF-	Module 1 ModRef[1023]		Module 1 ModRef[1022]		Module 1 ModRef[1021]		Module 1 ModRef[1020]		Module 1 ModRef[1019]		Module 1 ModRef[1018]		Module 1 ModRef[1017]		Module 1 ModRef[1016]	
0x00_100-	Module 2 ModRef[7]		Module 2 ModRef[6]		Module 2 ModRef[5]		Module 2 ModRef[4]		Module 2 ModRef[3]		Module 2 ModRef[2]		Module 2 ModRef[1]		Module 2 ModRef[0]	

Table 74: Program Linkage Table (Maximum Configuration: 65_536 Modules / 2560kB)

0x00_101-	Module 2 ModRef[15]	Module 2 ModRef[14]	Module 2 ModRef[13]	Module 2 ModRef[12]	Module 2 ModRef[11]	Module 2 ModRef[10]	Module 2 ModRef[9]	Module 2 ModRef[8]
0x00_102-	Module 2 ModRef[23]	Module 2 ModRef[22]	Module 2 ModRef[21]	Module 2 ModRef[20]	Module 2 ModRef[19]	Module 2 ModRef[18]	Module 2 ModRef[17]	Module 2 ModRef[16]
⋮	⋮							
0x00_17E-	Module 2 ModRef[1015]	Module 2 ModRef[1014]	Module 2 ModRef[1013]	Module 2 ModRef[1012]	Module 2 ModRef[1011]	Module 2 ModRef[1010]	Module 2 ModRef[1009]	Module 2 ModRef[1008]
0x00_17F-	Module 2 ModRef[1023]	Module 2 ModRef[1022]	Module 2 ModRef[1021]	Module 2 ModRef[1020]	Module 2 ModRef[1019]	Module 2 ModRef[1018]	Module 2 ModRef[1017]	Module 2 ModRef[1016]
⋮	⋮							
0x1F_F00-	Module 1022 ModRef[7]	Module 1022 ModRef[6]	Module 1022 ModRef[5]	Module 1022 ModRef[4]	Module 1022 ModRef[3]	Module 1022 ModRef[2]	Module 1022 ModRef[1]	Module 1022 ModRef[0]
0x1F_F01-	Module 1022 ModRef[15]	Module 1022 ModRef[14]	Module 1022 ModRef[13]	Module 1022 ModRef[12]	Module 1022 ModRef[11]	Module 1022 ModRef[10]	Module 1022 ModRef[9]	Module 1022 ModRef[8]
0x1F_F02-	Module 1022 ModRef[23]	Module 1022 ModRef[22]	Module 1022 ModRef[21]	Module 1022 ModRef[20]	Module 1022 ModRef[19]	Module 1022 ModRef[18]	Module 1022 ModRef[17]	Module 1022 ModRef[16]
⋮	⋮							
0x1F_F7E-	Module 1022 ModRef[1015]	Module 1022 ModRef[1014]	Module 1022 ModRef[1013]	Module 1022 ModRef[1012]	Module 1022 ModRef[1011]	Module 1022 ModRef[1010]	Module 1022 ModRef[1009]	Module 1022 ModRef[1008]
0x1F_F7F-	Module 1022 ModRef[1023]	Module 1022 ModRef[1022]	Module 1022 ModRef[1021]	Module 1022 ModRef[1020]	Module 1022 ModRef[1019]	Module 1022 ModRef[1018]	Module 1022 ModRef[1017]	Module 1022 ModRef[1016]
0x1F_F80-	Module 1023 ModRef[7]	Module 1023 ModRef[6]	Module 1023 ModRef[5]	Module 1023 ModRef[4]	Module 1023 ModRef[3]	Module 1023 ModRef[2]	Module 1023 ModRef[1]	Module 1023 ModRef[0]
0x1F_F81-	Module 1023 ModRef[15]	Module 1023 ModRef[14]	Module 1023 ModRef[13]	Module 1023 ModRef[12]	Module 1023 ModuleRef[11]	Module 1023 ModRef[10]	Module 1023 ModRef[9]	Module 1023 ModRef[8]

Table 74: Program Linkage Table (Maximum Configuration: 65_536 Modules / 2560kB)

0x1F_F82-	Module 1023 ModRef[23]	Module 1023 ModRef[22]	Module 1023 ModRef[21]	Module 1023 ModRef[20]	Module 1023 ModRef[19]	Module 1023 ModRef[18]	Module 1023 ModRef[17]	Module 1023 ModRef[16]
⋮	⋮							
0x1F_FFE-	Module 1023 ModRef[1015]	Module 1023 ModRef[1014]	Module 1023 ModRef[1013]	Module 1023 ModRef[1012]	Module 1023 ModRef[1011]	Module 1023 ModRef[1010]	Module 1023 ModRef[1009]	Module 1023 ModRef[1008]
0x1F_FFF-	Module 1023 ModRef[1023]	Module 1023 ModRef[1022]	Module 1023 ModRef[1021]	Module 1023 ModRef[1020]	Module 1023 ModRef[1019]	Module 1023 ModRef[1018]	Module 1023 ModRef[1017]	Module 1023 ModRef[1016]
0x20_000-	System (Module, Version) Translation[3]		System (Module, Version) Translation[2]		System (Module, Version) Translation[1]		System (Module, Version) Translation[0]	
0x20_001-	System (Module, Version) Translation[7]		System (Module, Version) Translation[6]		System (Module, Version) Translation[5]		System (Module, Version) Translation[4]	
0x20_002-	System (Module, Version) Translation[11]		System (Module, Version) Translation[10]		System (Module, Version) Translation[9]		System (Module, Version) Translation[8]	
⋮	⋮							
0x27_FFE-	System (Module, Version) Translation[65_531]		System (Module, Version) Translation[65_530]		System (Module, Version) Translation[65_529]		System (Module, Version) Translation[65_528]	
0x27_FFF-	System (Module, Version) Translation[65_535]		System (Module, Version) Translation[65_534]		System (Module, Version) Translation[65_533]		System (Module, Version) Translation[65_532]	

Table 75: Program Linkage Table (Example Configuration: 64 Linking Modules + 64 Leaf Modules / 8704 Bytes)

	Byte															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x000-	Module 0 Module[15:0] (note: when Module count is less than 256, a single byte index is sufficient)															
0x001-	Module 0 ModuleReference[31: 16]															
0x002-	Module 0 ModuleReference[47: 32]															
⋮	⋮															
0x007-	Module 0 ModuleReference[111: 96]															
0x007-	Module 0 ModuleReference[127:112]															
0x008-	Module 1 ModuleReference[15: 0]															
0x008-	Module 1 ModuleReference[31: 16]															
0x008-	Module 1 ModuleReference[47: 32]															
⋮	⋮															
0x00F-	Module 1 ModuleReference[111: 96]															
0x00F-	Module 1 ModuleReference[127:112]															
0x010-	Module 2 ModuleReference[15: 0]															
0x010-	Module 2 ModuleReference[31: 16]															
0x010-	Module 2 ModuleReference[47: 32]															
⋮	⋮															
0x017-	Module 2 ModuleReference[111: 96]															
0x017-	Module 2 ModuleReference[127:112]															
⋮	⋮															

Table 75: Program Linkage Table (Example Configuration: 64 Linking Modules + 64 Leaf Modules / 8704 Bytes)

0x1E0-	Module 62 ModuleReference[15: 0]			
0x1E1-	Module 62 ModuleReference[31: 16]			
0x1E2-	Module 62 ModuleReference[47: 32]			
⋮	⋮			
0x1EE-	Module 62 ModuleReference[111: 96]			
0x1EF-	Module 62 ModuleReference[127:112]			
0x1f0-	Module 63 ModuleReference[15: 0]			
0x1f1-	Module 63 ModuleReference[31: 16]			
0x1f2-	Module 63 ModuleReference[47: 32]			
⋮	⋮			
0x1FE-	Module 63 ModuleReference[111: 96]			
0x1FF-	Module 63 ModuleReference[127:112]			
0x200-	System (Module, Version) Translation[3]	System (Module, Version) Translation[2]	System (Module, Version) Translation[1]	System (Module, Version) Translation[0]
0x201-	System (Module, Version) Translation[7]	System (Module, Version) Translation[6]	System (Module, Version) Translation[5]	System (Module, Version) Translation[4]
0x202-	System (Module, Version) Translation[11]	System (Module, Version) Translation[10]	System (Module, Version) Translation[9]	System (Module, Version) Translation[8]
⋮	⋮			
0x21E-	System (Module, Version) Translation[123]	System (Module, Version) Translation[122]	System (Module, Version) Translation[121]	System (Module, Version) Translation[120]
0x21F-	System (Module, Version) Translation[127]	System (Module, Version) Translation[126]	System (Module, Version) Translation[125]	System (Module, Version) Translation[124]

Table 76: Program Linkage Table (Small Configuration: 8 Linking Modules + 32 Leaf Modules / 2304 Bytes)

	Byte															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x000-	Module 0 Module[15:0] (8 < maximum-external-module-references <= 16)															
0x001-	Module 1 ModuleReference[15:0]															
0x002-	Module 2 ModuleReference[15:0]															
⋮	⋮															
0x006-	Module 6 ModuleReference[15:0]															
0x007-	Module 7 ModuleReference[15:0]															
0x080-	System (Module, Version) Translation[3]				System (Module, Version) Translation[2]				System (Module, Version) Translation[1]				System (Module, Version) Translation[0]			
0x081-	System (Module, Version) Translation[7]				System (Module, Version) Translation[6]				System (Module, Version) Translation[5]				System (Module, Version) Translation[4]			
0x082-	System (Module, Version) Translation[11]				System (Module, Version) Translation[10]				System (Module, Version) Translation[9]				System (Module, Version) Translation[8]			
⋮	⋮															
0x08E-	System (Module, Version) Translation[59]				System (Module, Version) Translation[58]				System (Module, Version) Translation[57]				System (Module, Version) Translation[56]			
0x08F-	System (Module, Version) Translation[63]				System (Module, Version) Translation[62]				System (Module, Version) Translation[61]				System (Module, Version) Translation[60]			

9.4.1 s

Table 77: Data System Reference (DSR) Format

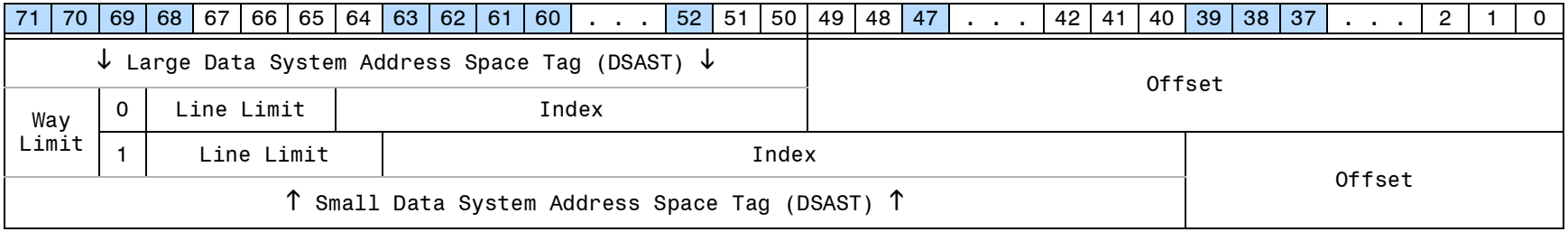


Table 78: Data System Reference (DSR) Field Descriptions

Bit Range	Field Name	Field Description
39:0 / 49:0	Offset	This field contains the Offset within the indicated DSAST of the associated reference. A Offset is 40 or 50 bits in size, depending on the size model of the associated data object (controlled by DSR bit 63). A large (50-bit) offset does not imply that the associated data object requires the larger offset's addressing range.
71:40 / 71:50	Data System Address Space Tag (DSAST)	Small data objects have a 32-bit DSAST and large data objects have a 22-bit DSAST. In both cases the DSAST contains 4 sub-fields, that in concert, represent the DSAST. The variable sized DSAST value is its associated data object's system-wide identifier. Almost 30% of the values within the DSAST's ranges are not valid identifiers (the invalid DSAST values are reserved or used for alternate DSR encodings). Two of the sub-fields impact the associated data object's cache placement, the third is a flag indicates whether the DSR references an LDO or an SDO, and the fourth is purely an identifying index, however, even the cache placement limiters are useful data object identifiers.
63:40 / 63:50	DSAST: Index	This field contains a 24-bit or 15-bit index value, that in concert with the cache placement restrictions (Line Limit and Way Limit) constitutes a 33-bit or 23-bit DSAST. Most DSASTs are usefully restricted either as a statement regarding their size or their relative importance. This field, independent of cache line placement restrictions, supports a concurrent, system-wide minimum (cache line placement restrictions act as Index multipliers) of $16*1024^2$ small DSASTs (1tB Offset range) and 32*1024 large DSASTs (1pB Offset range).

Table 79: Large Data Process Address Space Tag Translation Table Entry Format

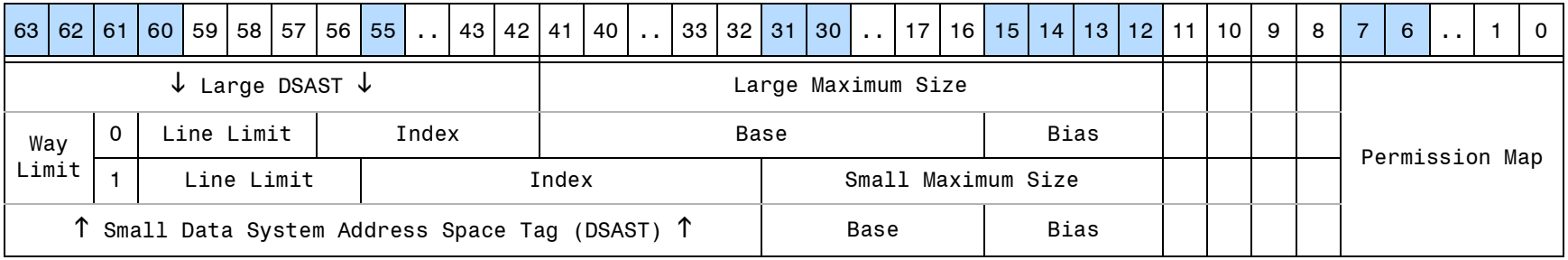


Table 80: GAST Control Structure Field Descriptions

Bit Range	Field Name	Field Description
7:0	Permission Map	This field contains a set of bit-mapped capability tags for the associated DPAS. Most of the bits are presently undefined, but the set of defined capability flags include:
		0 = The associated data object is null **1** = Read. **2** = Modify. **3** = Append. **4** = **5** = **6** = The associated DSAST cannot shrink via this reference. **7** = The associated DPAST cannot grow the size of the associated DSAS.
11:8	RESERVED	
41:12	Large Maximum Size	This field has a standard Base(26),Bias(4) interpretation: `(~Bias) ? (0x400_0000 | Base) << Bias : Base`. This field represents the maximum allowed size of the associated DPAS with decreasing resolution that starts at 1024 bytes. A field value of (0,15) is interpreted as an unlimited sizes DSAST.
63:52	Large DSAST	This field contains the Large DSAST index translation for the associated DPAST table index.
31:12	Small Maximum Size	This field has a standard Base(16),Bias(4) interpretation: `(~Bias) ? (0x10000 | Base) << Bias : Base`. This field represents the maximum allowed size of the associated DPAS with decreasing resolution that starts at 1024 bytes. A field value of (0,15) is interpreted as an unlimited sizes DSAST.
63:32	Small DSAST	This field contains the Small DSAST index translation for the associated DPAST table index.

Table 81: Context Address Space Format

	Byte															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00_000-	Program Counter								Processor Status Word							
0x00_001-	Processor Status Word Extension															
0x00_002-																
0x00_003-																
0x00_004-	System High MN Configuration															
0x00_005-	System Low MN Configuration															
0x00_006-	User High MN Configuration															
0x00_007-	User Low MN Configuration															
0x00_008-																
0x00_03E-																
0x00_03F-																
0x00_040-																
0x00_041-																
0x00_07E-																
0x00_07F-																
0x00_080-	Large DPAST[1]-to-DSAST								Large DPAST[0]-to-DSAST							
0x00_081-	Large DPAST[3]-to-DSAST								Large DPAST[2]-to-DSAST							

Table 81: Context Address Space Format

0x00_082-	Large DPAST[5]-to-DSAST								Large DPAST[4]-to-DSAST							
⋮	⋮								⋮							
0x00_0DE-	Large DPAST[189]-to-DSAST								Large DPAST[188]-to-DSAST							
0x00_0DF-	Large DPAST[191]-to-DSAST								Large DPAST[190]-to-DSAST							
0x00_0E0-	Source PSNE[3]-to-SSNE				Source PSNE[2]-to-SSNE				Source PSNE[1]-to-SSNE				Source PSNE[0]-to-SSNE			
0x00_0E1-	Source PSNE[7]-to-SSNE				Source PSNE[6]-to-SSNE				Source PSNE[5]-to-SSNE				Source PSNE[4]-to-SSNE			
0x00_0E2-	Source PSNE[11]-to-SSNE				Source PSNE[10]-to-SSNE				Source PSNE[9]-to-SSNE				Source PSNE[8]-to-SSNE			
0x00_0E3-	Source PSNE[15]-to-SSNE				Source PSNE[14]-to-SSNE				Source PSNE[13]-to-SSNE				Source PSNE[12]-to-SSNE			
0x00_0E4-	Target PSNE[3]-to-SSNE				Target PSNE[2]-to-SSNE				Target PSNE[1]-to-SSNE				Target PSNE[0]-to-SSNE			
0x00_0E5-	Target PSNE[7]-to-SSNE				Target PSNE[6]-to-SSNE				Target PSNE[5]-to-SSNE				Target PSNE[4]-to-SSNE			
0x00_0E6-	Target PSNE[11]-to-SSNE				Target PSNE[10]-to-SSNE				Target PSNE[9]-to-SSNE				Target PSNE[8]-to-SSNE			
0x00_0E7-	Target PSNE[15]-to-SSNE				Target PSNE[14]-to-SSNE				Target PSNE[13]-to-SSNE				Target PSNE[12]-to-SSNE			
0x00_0E8-	PSAE[3]-to-SSNE				PSAE[2]-to-SSNE				PSAE[1]-to-SSNE				PSAE[0]-to-SSNE			
0x00_0E9-	PSAE[7]-to-SSNE				PSAE[6]-to-SSNE				PSAE[5]-to-SSNE				PSAE[4]-to-SSNE			
0x00_0EA-	PSAE[11]-to-SSNE				PSAE[10]-to-SSNE				PSAE[9]-to-SSNE				PSAE[8]-to-SSNE			
0x00_0EB-	PSAE[15]-to-SSNE				PSAE[14]-to-SSNE				PSAE[13]-to-SSNE				PSAE[12]-to-SSNE			
0x00_0EC-																
0x00_0ED-																
0x00_0EE-																
0x00_0EF-																
0x00_0F0-																
0x00_0F1-																

Table 81: Context Address Space Format

⋮	⋮															
0x00_0FE-																
0x00_0FF-																
0x00_100-																
0x00_101-																
0x09_000-																
0x09_001-																
0x09_FFE-																
0x09_FFF-																
0x0A_000-																
0x0A_001-																
0x0A_FFE-																
0x0A_FFF-																
0x0B_000-																
0x0B_001-																

Table 81: Context Address Space Format

0x0B_FFE-																
0x0B_FFF-																
0x0C_000-																
0x0C_001-																
0x0C_FFE-																
0x0C_FFF-																
0x0D_000-																
0x0D_001-																
0x0D_FFE-																
0x0D_FFF-																
0x0E_000-	Register 1								Register 0							
0x0E_001-	Register 3								Register 2							
⋮	⋮								⋮							
0x0F_FFF-	Register 16_381								Register 16_380							
0x0F_FFF-	Register 16_383								Register 16_382							
0x10_000-	Private Local Store 1 Mibibyte Maximum Size															
0x10_001-																
⋮																
0x1F_FFE-																
0x1F_FFF-																
0x20_000-	High Priority, System Messaged Notification Queue Entry[0]															

Table 81: Context Address Space Format

0x20_001-	High Priority, System Messaged Notification Queue Entry[1]	
⋮	⋮	
0x27_FFE-	High Priority, System Messaged Notification Queue Entry[32_766]	
0x27_FFF-	High Priority, System Messaged Notification Queue Entry[32_767]	
0x28_000-	Low Priority, System Messaged Notification Queue Entry[0]	
0x28_001-	Low Priority, System Messaged Notification Queue Entry[1]	
⋮	⋮	
0x2F_FFE-	Low Priority, System Messaged Notification Queue Entry[32_766]	
0x2F_FFF-	Low Priority, System Messaged Notification Queue Entry[32_767]	
0x30_000-	High Priority, User Messaged Notification Queue Entry[0]	
0x30_001-	High Priority, User Messaged Notification Queue Entry[1]	
⋮	⋮	
0x37_FFE-	High Priority, User Messaged Notification Queue Entry[32_766]	
0x37_FFF-	High Priority, User Messaged Notification Queue Entry[32_767]	
0x38_000-	Low Priority, User Messaged Notification Queue Entry[0]	
0x38_001-	Low Priority, User Messaged Notification Queue Entry[1]	
⋮	⋮	
0x3F_FFE-	Low Priority, User Messaged Notification Queue Entry[32_766]	
0x3F_FFF-	Low Priority, User Messaged Notification Queue Entry[32_767]	
0x40_000-	Source NPAST[1]-to-DSAST	Source NPAST[0]-to-NSAST
0x40_001-	Source NPAST[3]-to-DSAST	Source NPAST[2]-to-NSAST
⋮	⋮	

Table 81: Context Address Space Format

0x4F_FFE-	Source NPAST[65_533]-to-DSAST	Source NPAST[65_532]-to-NSAST
0x4F_FFF-	Source NPAST[65_535]-to-DSAST	Source NPAST[65_534]-to-NSAST
0x50_000-	Target NPAST[1]-to-DSAST	Target NPAST[0]-to-NSAST
0x50_001-	Target NPAST[3]-to-DSAST	Target NPAST[2]-to-NSAST
⋮	⋮	⋮
0x5F_FFE-	Target NPAST[65_533]-to-DSAST	Target NPAST[65_532]-to-NSAST
0x5F_FFF-	Target NPAST[65_535]-to-DSAST	Target NPAST[65_534]-to-NSAST
0x60_000-	APAST[1]-to-DSAST	APAST[0]-to-NSAST
0x60_001-	APAST[3]-to-DSAST	APAST[2]-to-NSAST
⋮	⋮	⋮
0x6F_FFE-	APAST[65_533]-to-DSAST	APAST[65_532]-to-NSAST
0x6F_FFF-	APAST[65_535]-to-DSAST	APAST[65_534]-to-NSAST
0x70_000-	Small DPAST[1]-to-DSAST	Small DPAST[0]-to-DSAST
0x70_001-	Small DPAST[3]-to-DSAST	Small DPAST[2]-to-DSAST
⋮	⋮	⋮
0x7F_FFE-	Small DPAST[65_533]-to-DSAST	Small DPAST[65_532]-to-DSAST
0x7F_FFF-	Small DPAST[65_535]-to-DSAST	Small DPAST[65_534]-to-DSAST
0x80_000-		
0x80_001-		
⋮	⋮	⋮
0xFF_FFE-		
0xFF_FFF-		

•

Table 82: Context Address Space (CAS) Format

Byte	3	2	1	0
0				
8				
16				
⋮				
0x3_FFFE				
0x3_FFFF				
0x4_0000				
0x4_0004				
0x4_0008				
⋮				
0x4_FFF8				
0x4_FFFC				
0x5_0000	MN Access Map [63: 0] 22-bit field; 4mb/64kw (64*1024*64)			
0x5_0001	MN Access Map [127: 64]			
0x5_0002	MN Access Map [191:128]			
⋮	⋮			
0x5_3FFE	MN Access Map [1_048_511:1_048_448]			

Table 82: Context Address Space (CAS) Format

0x5_3FFF	MN Access Map [1_048_575:1_048_512]
0x5_4000	Code Module 0 Configuration Word
0x5_4001	Code Module 1 Configuration Word
0x5_4002	Code Module 2 Configuration Word
⋮	⋮
0x5_403E	Code Module 62 Configuration Word
0x5_403F	Code Module 63 Configuration Word
0x5_4040	CAST 2 (Root) | CAST 1 (Parent) | CAST 0 (Self)
0x5_4041	CAST 3 (Message) | CAST 4 (Last Child) | CAST 5 (First Child)
0x5_4042	… | CAST 7 (Previous Peer) | CAST 6 (Next Peer)
0x5_4043	… | Secondary Spawn Count | Primary Spawn Limit
0x5_4044	… | Secondary Spawn Limit | Primary Spawn Count
0x5_4045	… | … | …
0x5_4046	… | … | …
0x5_4047	… | … | …
⋮	⋮
0x5_5FF0	… | … | …

Table 82: Context Address Space (CAS) Format

0x5_5FF1									...									...									...				
0x5_5FF2									...					Blocked List Previous CAST									Blocked List Next CAST							
0x5_5FF3									...					Event List Previous CAST									Event List Next CAST							
0x5_5FF4																															
0x5_5FF5																															
0x100_0000	Global Module Hash Table Index [0]																														
0x100_0008	Global Module Hash Table Index [1]																														
⋮	⋮																														
0x107_FFF0	Global Module Hash Table Index [65_534]																														
0x107_FFF8	Global Module Hash Table Index [65_535]																														