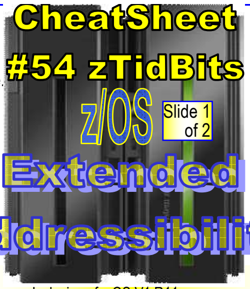
CheatSheet #54 zTidBits z/OS Extended Addressibility



Slide 1 of 2

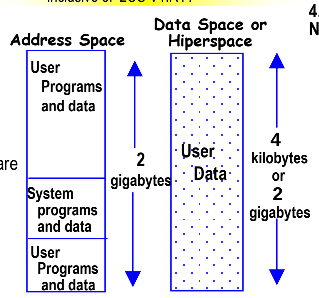
Inclusive of zOS V1.R11

* Two key areas of zOS growth and change are addressability and integrity.
- An address space, literally defined as the range of addresses available to a computer program, is like a programmer's map of the virtual storage available for code & data.
- An address space provides each programmer with access to all of the addresses available through the z architecture.
- Because it maps all of the available addresses, an address space includes system code and data as well as user code and data; thus, not all of the mapped addresses are available for user access. **NOTE:** This limit on user applications was a major reason for System/370 Extended Architecture (370-XA) and MVS/XA.

* Because the effective length of an address field expanded from 24 bits to 31 bits, the size of an address space expanded from 16 megabytes to 2 gigabytes. An MVS/XA address space is 128 times as big as an MVS/370 address space. A 2-gigabyte address space, however, does not, in and of itself, meet all of programmers' needs in an environment where processor speed continues to increase, where applications must support hundreds of users with instant response time requirements, and where businesses depend on quick access to huge amounts of information stored on DASD.

* With z/OS, the MVS address space expands to a size so vast that we need new terms to describe it.
- Each address space, called a 64-bit address space, is 16 exabytes in size; an exabyte is slightly more than one billion gigabytes.
- The new address space has logically 264 addresses and 8 billion times the size of the former 2-gigabyte address space that logically has 231 addresses.
  NOTE: The number is 16 with 18 zeros after it: 16,000,000,000,000,000,000 bytes, or 16 exabytes so if you are coding a new program that needs to store large amounts of data, a 64-bit address space might work for you.

* If, however you need more than a large address space, other extended addressability techniques meet that need.
- Extended addressability allows programmers to extend the power of applications through the use of additional address spaces or data-only spaces.
- The data-only spaces that are available for your programs are called data spaces and hiperspaces. These spaces are similar in that both are areas of virtual storage that your program can ask the system to create. Their size can be up to 2 gigabytes, as your program requests. Unlike an address space, a data space or hiperspace contains *only* user data; it does not contain system control blocks or common areas.
  NOTE: Program code can not run in a data space or a hiperspace.



* Both the architecture and the system protect the integrity of code and data within an address space.
- Various techniques, like storage protect key and supervisor state requirements, provide protection that is almost like a wall around an address space, and this wall is basically a good thing from the point of view of the work going on inside that individual address space.
- The programming techniques that provide extended addressability permit programs to break through but still preserve the wall that protects the address space. Whether your application is one that can use extended addressability depends on many factors.
  > One basic factor is the ***amount of central and auxiliary storage*** available at your installation to back up virtual storage.

* At the detail level, extended addressability can mean learning *new programming techniques*, or new ways of applying existing techniques.

* At a higher level, extended addressability can *open completely different solutions* to programming problems.
- These methods can become, conceptually, "a high-performance medium for application data".

* **Asynchronous cross memory communication** is a fancy way to describe scheduling an **SRB** which is a service request block that a task can schedule to request that some service take place in the *same* address space or *another* address space.
- Any data that the requesting task and the service share must be placed in common storage and serve as one way to overlap processing.
- A task schedules an SRB to perform a service, then continues with its work. When the service completes, it informs the task.
  NOTE: The timing, however, is asynchronous; the point when the SRB completes cannot be predicted.

* **Synchronous cross memory communication**, called cross memory, is both more complex and more flexible than scheduling an SRB.
- Cross memory requires the programmer to use zOS macros to establish a cross memory environment and once this environment is established, the authorized application can use assembler instructions to transfer control from one address space to another.

* **Data-in-virtual** enables you to map data into virtual storage but deal *only* with the portion of it that you need.
- The DIV macro provides the system services that manage the data object and enables you to map the object into virtual storage, create a window, and "view" through that window only the portion of the data object that you need.
- The system brings into central storage only the data that you actually reference.
- You can map a data-in-virtual object in either an address space, a data space, or a hiperspace.
- Mapping the object into a data space or hiperspace provides additional storage for the data; the size of the window is no longer restricted to the space available in an address space and provides additional isolation and integrity for the data, as well as more direct methods of sharing access to that data.
- Data-in-virtual is most useful for applications, such as BLOBs, that require large amounts of data but normally reference only small portions at any time.
- It requires that the source of the object be a VSAM linear data set on DASD (a permanent object) or a hiperspace (a temporary object) and is also useful for applications that require small amounts of data; data-in-virtual simplifies the way you access data by *avoiding the complexities of access methods*.

* The **virtual lookaside facility (VLF)** is a set of zOS services providing a high-performance alternate path method of retrieving named objects from DASD on behalf of many users and is designed primarily to improve the response time for such applications.
- VLF uses data spaces to hold data objects in virtual storage as an alternative to repeatedly retrieving the data from DASD so if you have an existing data retrieval application or are considering designing one, determine whether VLF can meet your needs.

* **Data spaces and hiperspaces** are data-only spaces that can hold up to 2 gigabytes of data and provide integrity and isolation for the data they contain in much the same way as address spaces provide integrity and isolation for the code and data they contain.
- They are **an extremely flexible solution** to problems related to accessing ***' large amounts of data '.***
- There are *two basic ways* to place data in a data space or a hiperspace.
  1. Through buffers in the program's address space. **2.** Using data-in-virtual services, a program can move data into a data space or hiperspace directly avoiding use of an address space's virtual storage as an intermediate buffer area.

NOTE: For hiperspaces, this second way **reduces the amount of I/O**.
- Programs that use data spaces run in AR ASC mode‡ using zOS macros to create, control, and delete data spaces.
- Assembler instructions executing in the address space directly manipulate data that resides in data spaces.
- Programs that use hiperspaces run in primary or AR ASC mode and use zOS macros to create, control, and delete hiperspaces.

NOTE: Programs cannot directly manipulate data in a hiperspace, but can use the macros to transfer data to and from the hiperspace for data manipulation.
- Hiperspaces provide **high-speed access** to large amounts of data; Hiperspace is the most efficient form of intermediate storage for DFSORT.

NOTE: DFSORT only uses *Hipersorting* when there is sufficient storage to back all the DFSORT Hiperspace data.

‡ In Access Register Address Space Control (AR ASC) mode, a program can move, compare, or perform operations on data in other address spaces or in data spaces.

* For storing data, zOS offers a program a choice of two kinds of virtual storage areas outside the program's address space: **data spaces** and **hiperspaces**.
- You must make these decisions: *Does my program need virtual storage outside the address space?*
  Which kind of virtual storage is appropriate for my program?

* Data spaces and hiperspaces are similar in both are areas of virtual storage that the program can ask the system to create
- They differ in the way your program accesses data in the two areas and what your program can do with these areas.
- Under certain conditions, virtual input/output (VIO) can be a better option than a data space or a hiperspace.

* Programs can use data spaces and hiperspaces to:
1. Obtain more virtual storage than a single address space gives a user.
2. Isolate data from other tasks in the address space.
   - Data in an address space is accessible to all programs executing in that address space. You might want to move some data to a data space or hiperspace for security or integrity reasons. You can restrict access to data in those spaces to one or several units of work.
3. Share data among programs that are executing in the same address space or different address spaces.
   - Instead of keeping the shared data in common areas, create a data space or hiperspace for the data you want your programs to share. Use this space as a way to separate your data logically by its own particular use.
4. Provide an area in which to map a data-in-virtual object.

NOTE: You can place all types of data in a data space or hiperspace, rather than in an address space or on DASD. Examples of such data include: tables, arrays, matrixes, data base buffers, temporary work files and even copies of permanent data sets.

Because data spaces and hiperspaces do not include system areas, the cost of creating and deleting them *is less* than that of an address space.

Base guidelines of data requirements for VIO, Data Spaces, and Hiperspaces

Question	***VIO***	***Data Space***	***Hiperspace***
Is the data in your program temporary?	Supports only temporary data	Support temporary data and permanent data (through DIV).	Supports temporary and permanent data thru DIV or data window services.
Is the data in your program sequential?	Using EXCP† supports both sequential and random access; however, random access requires more processor cycles.	Has no requirement.	Has no requirement.
Is the data arranged (or able to be organized) in 4KB blocks?	Has no requirement.	Has no requirement.	Requires that the data be accessed and referenced in 4KB increments, located on a 4KB boundary.
How difficult is it to modify existing programs that use I/O operations?	Requires no modification to existing programs that use an access method with EXCP. Either use storage management subsystem (SMS) to make global request to use VIO or use JCL for an individual program.	Programs must change to use system macros and access register. Through VLF, authorized programs can use data spaces to store and retrieve named objects. HLL programs can NOT use data spaces directly.	Programs must change to use system macros or data window services. HLL programs can NOT use hiperspaces directly. They can use hiperspaces through data window services.

* The main difference between data spaces and hiperspaces is the way a program references data.
- A program references data in a data space **directly**, in much the same way it references data in an address space.
- It addresses the data by the byte, manipulating, comparing, and performing arithmetic operations.
- The program uses the same instructions (such as load, compare, add, and move character) that it would use to access data in its own address space.

* zOS provides a system service, the **HSPSERV macro**, to transfer the data between an address space and a hiperspace in 4K byte blocks.

* Virtual input/output (VIO), like data spaces and hiperspaces, is designed to reduce the need for the processor to transfer data between DASD and central storage.

* You might have an existing program — a high-level-language (HLL) program — (or assembler) that you would like to change to use the performance benefits of VIO, data spaces, or hiperspaces.
What is the performance comparison? Data spaces and hiperspaces do *not* have the overhead of an access method and the device simulation of VIO; therefore, they require less processor time than VIO.
For more information See the *zOS Programming: Assembler Services Guide.*

When would you choose VIO over data spaces or hiperspaces? Use VIO when you want to improve the performance of an existing program, but you do not want to make large changes (simple to set up via JCL).
For information about how to use VIO, see *z/OS MVS JCL User's Guide*.

† EXecute Channel Program (EXCP) initiates the channel program's I/O operations.

See SLIDE#2: Using the 64bit Address Space

CheatSheet #54 zTidBits z/OS Extended Addressibility

Inclusive of zOS V1.R11

* Because of changes in the architecture that supports the Multiple Virtual Storage (MVS) operating system, there have been two different address spaces *prior* to the 64-bit address space.
- The address space of the 1970s began at address 0 and ended at 16 megabytes this address space provided **24-bit addresses**.
- In the early 1980s, XA (extended architecture) introduced an address space that began at address 0 and ended at two gigabytes that created this address space provided **31-bit addresses**.
* To maintain program compatibility, MVS provided two addressing modes (AMODEs):
  - Programs that run in AMODE 24 can use only the first 16 megabytes of the address space
  - Programs that run in AMODE 31 can use the entire 2 gigabytes.
* As of z/OS Release 1.2, the address space begins at address 0 and ends at 16 exabytes, an *incomprehensibly* high address and the architecture that creates this address space provides 64-bit addresses.
* The address space structure below the 2 gigabyte address has not changed; all programs in AMODE 24 and AMODE 31 continue to run without change.
  - In some fundamental ways, the address space is much the same as the XA address space.
  - In the 31-bit address space, a virtual **line** marks the 16-megabyte address.
* The 64-bit address space also includes the virtual line at the 16-megabyte address; additionally, it includes a second virtual line called **the bar** that marks the 2-gigabyte address.
  - The bar separates storage below the 2-gigabyte address, called **below the bar**, from storage above the 2-gigabyte address, called **above the bar**.
  - The area above the bar is intended for data; no programs run above the bar.
  - There is no area above the bar that is common to all address spaces (at this time), and no system control blocks exist above the bar.

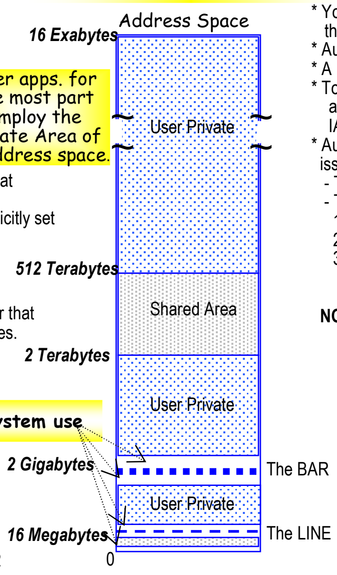
NOTE: IBM reserves an area of storage above the bar for special uses to be developed in the future.
- You can set a limit on how much virtual storage above the bar each address space can use.
  > This limit is called the **MEMLIMIT, but** If you do not set a MEMLIMIT, the system default is 2G, meaning that the address space can use up to 2G of virtual storage above the bar.
- If you want an address space to have access to more or less virtual storage above the bar, you need to explicitly set the MEMLIMIT to the limit you want.

NOTE: You can set an installation default MEMLIMIT through System Management Facility (SMF) or set a MEMLIMIT for a specific address space in the job control language (JCL) that creates the address space or by using exit IEFUSI.

* Figure on right shows a z/OS address space, including the line that marks the 16-megabyte address, the bar that marks the 2-gigabyte address, and the default shared area starting at 2 terabytes and ending at 512 terabytes.
* Before z/OS v1.3, all programs in AMODE 31 or AMODE 24 were unable to work with data above the bar.
  - To use virtual storage above the bar, a program must request storage above the bar, be in AMODE 64, and use the new z/Architecture assembler instructions.
* As of z/OS v1.5, the following enhancements for 64-bit virtual storage have been added:
  1. 64-bit shared memory support
  2. Multiple guard area support for private high virtual storage
  3. Default shared memory addressing area between 2 terabytes and 512 terabytes
* **Why would you use virtual storage above the BAR?**
  - The program needs more virtual storage than the first 2-gigabyte address space provides. Before z/OS 1.2, a program's need for storage beyond what the former 2-gigabyte address space provided was sometimes met by creating one or more data spaces or hiperspaces and then designing a memory management schema to keep track of the data in those spaces. Sometimes programs written before Zos1.2 used complex algorithms to manage storage, reallocate and reuse areas, and check storage availability.
  - With the 16-exabyte address space, these kinds of programming complexities are unnecessary because a program can potentially have as much virtual storage as it needs, while containing the data within the program's home address space.
* A good example of a programming model that can successfully take advantage of the 16-exabyte address space is a program that needs very large buffer pools and has typically used multiple data spaces and then managed them separately.
* **Memory management above the 2 GB bar** is organized as memory objects that a program creates.
  - A memory object is a contiguous range of virtual addresses that are allocated by programs as a number of application pages which are 1MB multiples on a 1MB boundary and programs continue to run and execute in the first 2GB of the address space.
* While there is no practical limit to the virtual storage above the bar, practical limits exist to the real storage frames and auxiliary storage slots that back the area.
  - To control the amount of real and auxiliary storage that an address space can use for memory objects at one time, your installation can establish an installation default.
  - MEMLIMIT that sets the total number of usable virtual pages above the bar for a single address space.
  - You set this default in two ways:
    1. On the MEMLIMIT parameter in the SMFPRMxx parmlib member
    2. Through the SET SMF or SETSMF commands.The default takes effect if a job does not specify MEMLIMIT on the JCL JOB or EXEC statement, or REGION=0 in the JCL;

NOTE: The MEMLIMIT specified in an IEFUSI exit routine overrides all other MEMLIMIT settings. If REGION=0 is specified in the JCL & the IEFUSI exit limits the REGION size but does not set MEMLIMIT, MEMLIMIT is defaulted to the REGION size above 16MB.
* The system enforces the MEMLIMIT when you issue the IARV64 GETSTOR and CHANGEGUARD services.
* When your unconditional request for new storage (either for a new memory object or for more usable storage in an existing memory object) causes the MEMLIMIT to be exceeded, the system abends the program. IBM recommends that programs use the COND parameter to make a conditional request and check the return code to make sure the storage is available.
* **Modifying MEMLIMIT** - If a SET SMF or SETSMF command changes the default MEMLIMIT (either the system default or the installation default) for an address space that is already created the following changes occur:
  1. If the command raises the current default MEMLIMIT, all address spaces whose MEMLIMIT values are set through SMF run with the higher default.
  2. If the command lowers the current default MEMLIMIT, all address spaces whose MEMLIMIT values are set through SMF keep their original (higher) system default.

See MEMLIMIT Decision Tree on right.



* Programs obtain storage above the bar in **"chunks"** of virtual storage called **memory objects**.
  - The system allocates a memory object as a number of virtual segments; each segment is a megabyte in size and begins on a megabyte boundary.
  - A memory object can be as large as the memory limits set by your installation and as small as one megabyte.
  - Other attributes of a memory object include the following characteristics:
    1. The storage key is defined by the program; for an unauthorized program, the storage key at the time of issuing the **IARV64 macro** is the program's PSW key
    2. You can specify whether you want the memory object to be fetch protected or not. There is no change key support for virtual storage above the bar.
    3. The owner of a private memory object is the TCB of the program that creates the private memory object, or a TCB to which the creating program assigns ownership. If an SRB creates a private memory object, the SRB must assign ownership of the private memory object to a task.
    4. A shared memory object is system owned. The cross-memory resource owner (CMRO) TCB of the address space owns the shared interest in the shared memory object.
* When a program creates a memory object, it provides an area in which the system returns the memory object's low address
* You can think of the address as the name of the memory object. After creating the memory object, the program can use the storage in the memory object as it used storage in the 2-gigabyte address space.
* Authorized programs can ask the system to pagefix areas of private memory objects, making pages unavailable for stealing
* A large page, or a 1–megabyte page, is a special-purpose performance feature for memory objects.
* To request large pages for backing the memory object, authorized programs and unauthorized programs with read authority to facility class IARRSM.LRGPAGES can specify the PAGEFRAMESIZE parameter when issuing the IARV64 GETSTOR macro.
* Authorized programs can also request large pages for common memory objects by using PAGEFRAMESIZE when issuing the IARV64 GETCOMMON macro. Large pages consume real storage and are "non-pageable".
  - The system programmer should carefully consider what applications are granted access to large pages.
  - The key factors to consider when granting access to large pages are as follows:
    1. Memory usage
    2. Page translation overhead for the workload
    3. Available large frame area where long running memory-intensive applications benefit most from using large pages.

NOTE: Short-lived processes with a small memory working set are not good candidates. The system programmer defines the amount of real storage that can be used for large pages with the LFAREA parameter. See IEASYSxx in SYS1.PARMLIB.

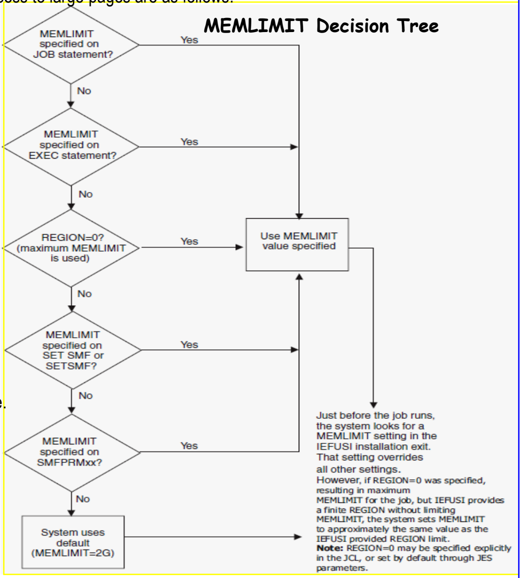
/* **WHY 31bit & not 32bit addressing** */
The 32bit tells z/OS what addressing mode a program can execute in.

	32bit = 1 (31bit mode)
AMODE31	Above the LINE
	32bit = 0 (24bit mode)
AMODE24	Below the LINE

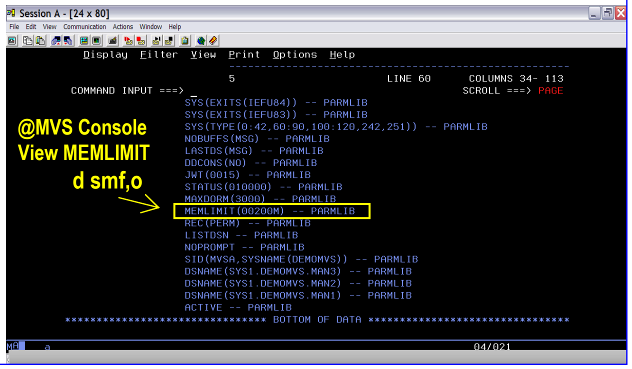
* A **subspace** is a specific range of storage in the private area of an address space, designed to limit the storage a program can reference.
* A program that is associated with a subspace cannot reference some of the private area storage outside of the subspace storage range; the storage is protected from the program. Whether a given range of private area storage is protected from a program associated with a subspace depends on whether the storage is:
  1. Eligible to be assigned to a subspace (or **"subspace-eligible"**)
  2. Assigned to a subspace
  3. Not eligible to be assigned to a subspace.

NOTE: Storage outside of the private area is not affected by subspaces. A program running in an address space can reference all of the storage associated with that address space. .

MEMLIMIT Decision Tree


MEMLIMIT specified on JOB statement?
Yes
No
MEMLIMIT specified on EXEC statement?
Yes
No
REGION=0? (maximum MEMLIMIT is used)
Yes
No
MEMLIMIT specified on SET SMF or SETSMF?
Yes
No
MEMLIMIT specified on SMFPRMxx?
Yes
No
Use MEMLIMIT value specified
System uses default (MEMLIMIT=2G)
Just before the job runs, the system looks for a MEMLIMIT setting in the IEFUSI installation exit. That setting overrides all other settings.
However, if REGION=0 was specified, resulting in maximum MEMLIMIT for the job, but IEFUSI provides a finite REGION without limiting MEMLIMIT, the system sets MEMLIMIT to approximately the same value as the IEFUSI provided REGION limit.
Note: REGION=0 may be specified explicitly in the JCL, or set by default through JES parameters.

This display indicates the status of SMF data sets or the SMF options in effect (message IEE967I).


Session A - [24 x 80]
File Edit View Communication Actions Window Help
@MVS Console View MEMLIMIT
d smf,o

```
          Display  Filter  View  Print  Options  Help
                       ------------------------------------------------------------
                       5                                  LINE 60       COLUMNS 34- 113
     COMMAND INPUT ===> _                                             SCROLL ===> PAGE
                 SYS(EXITS(IEFU84)) -- PARMLIB
                 SYS(EXITS(IEFU83)) -- PARMLIB
                 SYS(TYPE(0:42,60:90,100:120,242,251)) -- PARMLIB
                 NOBUFFS(MSG) -- PARMLIB
                 LASTDS(MSG) -- PARMLIB
                 DDCONS(NO) -- PARMLIB
                 JWT(0015) -- PARMLIB
                 STATUS(010000) -- PARMLIB
                 MAXDORM(3000) -- PARMLIB
                 MEMLIMIT(00200M) -- PARMLIB
                 REC(PERM) -- PARMLIB
                 LISTDSN -- PARMLIB
                 NOPROMPT -- PARMLIB
                 SID(MVSA,SYSNAME(DEMOMVS)) -- PARMLIB
                 DSNAME(SYS1.DEMOMVS.MAN3) -- PARMLIB
                 DSNAME(SYS1.DEMOMVS.MAN2) -- PARMLIB
                 DSNAME(SYS1.DEMOMVS.MAN1) -- PARMLIB
                 ACTIVE -- PARMLIB
     ******************************** BOTTOM OF DATA ********************************
MA    a                                                                04/021
```