IBM Enterprise Metal C for z/OS, V3.1
Version 3 Release 1

Optimization and Programming Guide

SC27-9402-00

Note

Before using this information and the product it supports, read the information in "Notices" on page 221.

This edition applies to Version 3 Release 1 of IBM® Enterprise Metal C for z/OS® (5655-MCE) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2019-10-23

Contents

About this document

This document contains reference information that is intended to help you understand the IBM Enterprise Metal C for z/OS compiler.

Typographical conventions

The following table explains the typographical conventions used in this document.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Commands, executable names, compiler options and pragma directives that contain lower-case letters.	The **metalc** invocation command invokes the Enterprise Metal C for z/OS compiler.
italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the *size* parameter if you return more than the *size* requested.
`monospace`	Programming keywords and library functions, compiler built-in functions, file and directory names, examples of program code, command strings, or user-defined names.	If one or two cases of a `switch` statement are typically executed much more frequently than other cases, break out those cases by handling them separately before the `switch` statement.

Softcopy documents

The Enterprise Metal C for z/OS publications are supplied in PDF format and available for download from the Enterprise Metal C for z/OS Knowledge Center home page (www.ibm.com/support/knowledgecenter/en/SSSHGK_3.1.0/com.ibm.metalc.v3r1.doc/welcome.html).

To read a PDF file, use the Adobe Reader. If you do not have the Adobe Reader, you can download it (subject to Adobe license terms) from the Adobe website (www.adobe.com).

You can also browse the documents on the World Wide Web by visiting the Enterprise Metal C for z/OS Knowledge Center home page (www.ibm.com/support/knowledgecenter/en/SSSHGK_3.1.0/com.ibm.metalc.v3r1.doc/welcome.html).

Where to find more information

For an overview of the information associated with z/OS, see .

Additional information on Enterprise Metal C for z/OS is available on the Marketplace page for Enterprise Metal C for z/OS (www.ibm.com/us-en/marketplace/xl-cpp-compiler-zos).

z/OS Basic Skills in IBM Knowledge Center

z/OS Basic Skills in IBM Knowledge Center is a Web-based information resource intended to help users learn the basic concepts of z/OS, the operating system that runs most of the IBM mainframe computers in use today. IBM Knowledge Center is designed to introduce a new generation of Information Technology professionals to basic concepts and help them prepare for a career as a z/OS professional, such as a z/OS system programmer.

Specifically, z/OS Basic Skills is intended to achieve the following objectives:

- Provide basic education and information about z/OS without charge
- Shorten the time it takes for people to become productive on the mainframe
- Make it easier for new people to learn z/OS.

z/OS Basic Skills in IBM Knowledge Center (www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zbasics/homepage.html) is available to all users (no login required).

Technical support

Additional technical support is available from the Enterprise Metal C for z/OS Support page (https://www.ibm.com/support/home/product/A032956W76367A04/IBM_Enterprise_Metal_C_for_z/OS). This page provides a portal with search capabilities to technical support FAQs and other support documents.

For the latest information about Enterprise Metal C for z/OS, visit Marketplace page for Enterprise Metal C for z/OS (www.ibm.com/us-en/marketplace/xl-cpp-compiler-zos).

If you cannot find what you need, you can e-mail:

compinfo@cn.ibm.com

How to send your comments to IBM

We appreciate your input on this documentation. Please provide us with any feedback that you have, including comments on the clarity, accuracy, or completeness of the information.

You can send an email to compinfo@cn.ibm.com and include the following information:

- Your name and address
- Your email address
- Your phone or fax number
- The publication title and order number:

  Enterprise Metal C for z/OS Optimization and Programming Guide
  SC27-9402-00
- The topic and page number or URL of the specific information to which your comment relates
- The text of your comment.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute the comments in any way appropriate without incurring any obligation to you.

IBM or any other organizations use the personal information that you supply to contact you only about the issues that you submit.

If you have a technical problem

If you have a technical problem, take one or more of the following actions:

- Visit the IBM Support Portal (support.ibm.com).
- Contact your IBM service representative.
- Call IBM technical support.

Part 1. Coding: Advanced Topics

This part contains the following coding topics:

Chapter 1. z/OS 64-bit environment

Implementation of the 64-bit environment has not changed the default behavior of the compiler; the default compilation environment is 32-bit, which is specified by the ILP32 compiler option.

The compiler changes the behavior of code only when compiling for the 64-bit environment, which is specified by the LP64 compiler option.

Differences between the ILP32 and LP64 environments

The ILP32 and LP64 environments are differentiated by:

- Addressing capability
- Data model

ILP32 and LP64 addressing capabilities

Table 2 on page 3 shows the differences in addressing capabilities that are available in each environment. 31-bit refers to the addressing mode, or AMODE. In Enterprise Metal C for z/OS, pointer sizes in this mode are always 4 bytes. In AMODE 31, 31 bits of the pointer are used to form the address, which is defined by the term *31-bit addressing mode*. Occasionally, we also use the term *32-bit mode*. Strictly speaking, 31-bit is an architectural characteristic referring to the addressing capability, while 32-bit is a programming language aspect referring to the data model. The latter is also referred to as ILP32 (int-long-pointer 32). When there is no ambiguity, we use the term *32-bit mode*.

Table 2. Comparison of ILP32 and LP64 addressing capabilities

ILP32 (32-bit environment)	**LP64 (64-bit environment)**
2 GB of address space	1 million TB of address space
31-bit execution mode	64-bit execution mode

ILP32 and LP64 data models and data type sizes

Table 3 on page 3 compares data models and data type sizes of ILP32 and LP64 environments.

Table 3. Comparison of ILP32 and LP64 data models

ILP32 (32-bit environment)	**LP64 (64-bit environment)**
Data model ILP32 (32-bit pointer)	Data model LP64 (64-bit pointer)
`int`, `long`, `ptr`, and `off_t` are all 32 bits (4 bytes) in size.	`int` is 32 bits in size. `long`, `ptr`, and `off_t` are all 64 bits (8 bytes) in size.

The 32-bit data model for Enterprise Metal C for z/OS is ILP32 plus `long long`. This data model uses the 4/4/4 data type size model and includes a `long long` type. Table 4 on page 3 compares the type sizes for the different models.

LP64 is the 64-bit data model chosen by the Aspen working group (formed by X/OPEN and a consortium of hardware vendors). LP64 is short for long-pointer 64. It is commonly referred to as the 4/8/8 data type size model and includes the integer/long/pointer type sizes, measured in bytes.

Table 4. ILP32 and LP64 type size comparisons for signed and unsigned data types

Data Type	**32-bit sizes (in bytes)**	**64-bit sizes (in bytes)**	**Remarks**
char	1	1	

Table 4. ILP32 and LP64 type size comparisons for signed and unsigned data types (continued)			
Data Type	**32-bit sizes (in bytes)**	**64-bit sizes (in bytes)**	**Remarks**
short	2	2	
int	4	4	
long	4	8	
long long	8	8	
float	4	4	
double	8	8	
long double	16	16	
pointer	4	8	
wchar_t	2	4	Other UNIX platforms usually have wchar_t 4 bytes for both 32-bit and 64-bit mode.
size_t	4	8	This is an unsigned type.
ptrdiff_t	4	8	This is a signed type.

Advantages and disadvantages of the LP64 environment

A major advantage of using a 64-bit environment is the increase in the virtual addressing space. A 64-bit program can handle large tables as arrays without putting temporary files in secondary storage. LP64 provides:

- 64-bit addressing with 8-byte pointers
- Large object support (8-byte longs)
- Backward compatibility (4-byte integers)

Note: Integers are the same size under the ILP32 and LP64 data models.

LP64 application performance and program size

You can use the 64-bit address space to dramatically improve the performance of applications that manipulate large amounts of data, whether the data is be created within the application or obtained from files. Generally, the performance gain comes from the fact that the 64-bit application can contain the data in its 64-bit address space (either created in data structures or mapped into memory), when it would not have fit into a 32-bit address space. The data would need to be multiple GBs in size or larger to show this benefit.

If the same source code is used to create a 32-bit and a 64-bit application, the 64-bit application is typically larger than the 32-bit application. The 64-bit application is unlikely to run faster than the 32-bit application unless it makes use of the larger 64-bit addressability. Because most C programs are pointer-intensive, a 64-bit application can be close to twice as large as a 32–bit application, depending on how many global pointers and longs are declared. That is why the appropriate choice is to create a 32-bit application, unless 64-bit addressability is required by the application or can be used to dramatically improve its performance.

Attention: Even though the address space is increased significantly, the amount of hardware physical memory is still limited by your installation. Data that is not immediately required by the program is subject to system paging. Programs that use large data tables therefore require a large amount of paging space. For example, if a program requires 3 GB of address space, the system must have 3 GB of paging space. 64-bit applications might require paging I/O tuning to accommodate the large data handling benefit.

LP64 restrictions

The following restrictions apply under LP64:

- The ILP32 statement `type=memory(hiperspace)` is treated as `type=memory` under LP64.

  Hiperspace memory files are treated as regular memory files in a 64-bit environment. All behavior is the same as for regular memory files.
- User-supplied buffers are ignored for all but UNIX file system files under LP64.

  References to user-supplied buffers are valid under ILP32 only.
- Under 64-bit data models, pointer sizes are always 64 bits.

  The C Standard does not provide a mechanism for specifying mixed pointer size. However, it might be necessary to specify the size of a pointer type to help migrate a 32-bit application (for example, when libraries share a common header between 32-bit and 64-bit applications).

Migrating applications from ILP32 to LP64

This section describes:

- When to migrate applications to LP64
- Pre-migration checklist
- Post-migration checklist

When to migrate applications to LP64

The LP64 strategy is to strike a balance between maximizing the robustness of 64-bit capabilities while minimizing the effort of migrating many programs.

Typically, a 32-bit application should be ported only if either of the following is true:

- It is required by a supporting utility
- It must have 64-bit addressability

This is because:

- Porting programs to a 64-bit environment presents a modest technical effort where good coding practices are used. Poor coding practices greatly increase the programming effort.
- There is no clear performance advantage to recompiling an existing 32-bit program in 64-bit mode. In fact, a small slowdown is possible. This is due to:
  - An increase in module size because instructions are larger
  - An increase in size of the writable static area (WSA) and the stack because pointers and longs are larger
  - Issues related to runtime requirements

Checklist for ILP32-to-LP64 pre-migration activities

Use the following checklist before migrating an application from ILP32 to LP64. After migration, test the code and confirm that its behavior is the same under LP64 as it was under ILP32. If you see any difference, debug the code and use the checklist again.

1. Search the source code for patterns that might indicate migration issues. These include:
   - 0xffffffff
   - 2147483647
2. Verify that all functions are properly prototyped.

   Note: The C compiler assumes that an unprototyped function returns the `int` type. This might cause undesirable behavior under LP64 while remaining undetectable under ILP32.

3. Examine all types to determine whether the types should be 4-byte or 8-byte.
   - For system types, the type will be the appropriate size for use with library/system calls.
   - For user-defined types:
     - 4-byte types should be defined based on int or unsigned int or some system type that is 4 bytes long under LP64.
     - 8-byte types should be defined based on long or unsigned long or some system type that is 8 bytes long.
4. Change all types to the chosen type.

   Note: When doing so, examine all arithmetic calculations to make sure that expansion and truncation of data values is done appropriately. Make sure that no assumption is made that pointer values fit into integer types.
5. Use the INFO compiler option to identify the following potential problems:
   - Functions not prototyped - Function prototypes allow the compiler to check for mismatched parameters.
   - Functions not prototyped - Return parameter mis-matched, especially when the code expects a pointer. (For example, `malloc` and `family`)
   - Assignment of a `long` or a pointer to an `int` - This type of assignment might cause truncation. Even assignments with an explicit cast will be flagged.
   - Assignment of an int to a pointer - If the pointer is referenced it might be invalid.

Checklist for ILP32-to-LP64 post-migration activities

After migrating a program, test the code and confirm that its behavior is the same under LP64 as it was under ILP32. Use the following checklist to test the code. If you see any difference, debug the code and use the pre-migration checklist again.

1. Verify that all output produced is contained in the 4-byte range.

   If this is not possible, then any other application using this data needs to be ported to LP64 or, at least, be made 8-byte-aware.
2. Verify that any user-provided process containing the `wchar_t` type definition did not produce unexpected results.

   UNIX `wchar_t` data types are typically defined as four bytes under both 32-bit and 64-bit environments. The size difference applies to the ILP32 model, not the LP64 model. The new environment was an opportunity to increase the size for future development. Because `wchar_t` is a type definition, user-provided methods are a likely problem area. A carefully-written application should not require changes.

Using compiler diagnostics to ensure portability of code

This section describes the following information:

- Using the INFO option to ensure that numbers are suffixed
- Using the WARN64 option to identify potential portability problems

Using the INFO option to ensure that numbers are suffixed

The INFO compiler option provides general diagnostics about program code and is not specific to migrations from ILP32 to LP64. Before migrating, use the appropriate option to ensure that the following items have been expunged from the code:

- Functions not prototyped - Function prototypes allow the compiler to check for mismatched parameters.

- Functions not prototyped - Return parameter mis-matched, especially when the code expects a pointer. (For example, `malloc` and `family`)
- Assignment of a `long` or a pointer to an `int` - This type of assignment could cause truncation. Even assignments with an explicit cast will be flagged.
- Assignment of an `int` to a pointer - If the pointer is referenced it might be invalid.

Using the WARN64 option to identify potential portability problems

Under ILP32, both `int` and `long` data types are 32 bits in size. Because of this coincidence, these types might have been used interchangeably. As shown in Table 4 on page 3, the data type `long` is 8 bytes in length under LP64.

A general guideline is to review the existing use of `long` data types throughout the source code. If the values to be held in such variables, fields, and parameters will fit in the range of $[-2^{31}...2^{31}-1]$ or $[0...2^{32}-1]$, then it is probably best to use `int` or `unsigned int` instead. Also, review the use of the `size_t` type (used in many subroutines), since its type is defined as `unsigned long`.

When you migrate a program from ILP32 to LP64, the data model differences might result in unexpected behavior at execution time. Under LP64, the size of pointers and `long` data types are 8 bytes, which can lead to conversion or truncation problems. The `WARN64` option can be used to detect these portability errors.

The `WARN64` option provides general diagnostics about program code that might behave differently under ILP32 and LP64. However the checking is not exhaustive. Use `WARN64` to look for potential migration problems, such as the following common problems:

- Truncation due to explicit or implicit conversion of `long` types into `int` types
- Unexpected results due to explicit or implicit conversion of `int` types into `long` types
- Invalid memory references due to explicit conversion by cast operations of pointer types into `int` types
- Invalid memory references due to explicit conversion by cast operations of `int` types into pointer types
- Problems due to explicit or implicit conversion of constants into `long` types
- Problems due to explicit or implicit conversion by cast operations of constants into pointer types

There are a few problems that `WARN64` cannot find. For example, unions that use `long` data types or pointers that work under ILP32 might not work under LP64 .

```
union {
   int *p;    /* 32 bits / 64 bits */
   int i;       /* 32 bits / 32 bits */
};

union {
   double d;      /* 64 bits / 64 bits */
   long l[2];       /* 64 bits / 128 bits */
};
```

ILP32-to-LP64 portability issues

Before migrating applications, consider the following:

- The sizes of the `long`, pointer, and `wchar_t` types are different under LP64 than they are under ILP32. You must check application behavior, especially if the logic depends on data size.
- Data model differences can result in unexpected behavior at execution time. Under LP64, the size of pointers and `long` data type are 8 bytes long. This can lead to conversion or truncation problems.

  Note: You can us the `WARN64` option to help detect these portability errors. See "Using the WARN64 option to identify potential portability problems" on page 7.

- A migration issue can exist if the program assumes that `int`, `long`, and pointer type are all the same size. The number of cases where program logic relies on this assumption varies from application to application, depending on the coding style and functionality of the application.

  Note: Most unexpected behaviors occur at the limits of a type's value range.
- 32-bit applications that rely implicitly on internal data representations (for example, those that cast a float pointer to an integer pointer, then manipulate the bit patterns directly and encode such knowledge directly into the program logic) can be difficult to migrate. In this case, certain assumptions are made about the internal structure of a float representation and the size of int.
- Code must be checked to ensure that any shifting and masking operations that manipulate long integers still work properly with a 64-bit long.
- Input and output file dependencies are relevant when you migrate an application that is in the middle of a pipeline of applications, where each application reads the previous application's output as input, and then passes its output to the next application in the pipe. Before migrating one of these applications to a 64-bit environment, you must verify that the output will not produce values outside of the 32-bit range. Typically, once an application is ported to a 64-bit environment, all downstream applications (that is, any application that depends on output from the ported application) must be ported to a 64-bit environment.
- Extending functions is sometimes included as part of a migration project to exploit the benefit and to justify the cost of migrating to a 64-bit environment. You might have to change code for using expanded limits after extending functions.

IPA(LINK) option and exploitation of 64-bit virtual memory

IPA(LINK) makes use of 64-bit virtual memory, which will cause a compiler ABEND if there is insufficient storage. The default MEMLIMIT system parameter size in the SMFPRMx parmlib member should be at least 3000 MB. The default MEMLIMIT value takes effect whenever the job does not specify one of the following:

- MEMLIMIT in the JCL JOB or EXEC statement
- REGION=0 in the JCL

Note: The MEMLIMIT value specified in an IEFUSI exit routine overrides all other MEMLIMIT settings.

The z/OS UNIX System Services **`ulimit`** command can be used to set the MEMLIMIT default. For information, see . For additional information about the MEMLIMIT system parameter, see .

The MTCI and MTCIA cataloged procedures, which are used for IPA Link, contain the variable IMEMLIM, which can be used to override the default MEMLIMIT value.

Potential changes in structure size and alignment

The LP64 specification changes the size and alignment of certain structure elements, which affects the size of the structure itself. In general, all structures that use long integers and pointers must be checked for size and alignment dependencies.

It is not possible to share a data structure between 32-bit and 64-bit processes, unless the structure is devoid of pointer and long types. Unions that attempt to share `long` and `int` types (or overlay pointers onto `int` types) will be aligned differently or will be corrupted.

Note: The issue of changing structure size and alignment should not be a problem unless the program makes assumptions about the size and/or composition of structures.

z/OS basic rule of alignment

The basic rule of alignment in z/OS is that a data structure is aligned in accordance with its size and the strictest alignment requirement for its largest member. An 8-byte alignment is more stringent than a 4-byte alignment. In other words, members that can be placed on a 4-byte boundary can also be placed on an 8-byte boundary, but not vice versa.

Note: The only exception is a `long double`, which is always aligned on an 8-byte boundary.

You can satisfy the rule of alignment by inserting pad members both between members and at the end of a structure, so that the overall size of the structure is a multiple of the structure's alignment.

Examples of structure alignment differences under ILP32 and LP64

This section provides examples of three structures that illustrate the impact of the ILP32 and LP64 programming environments on structure size and alignment.

In accordance with the z/OS rule of alignment (see "z/OS basic rule of alignment" on page 8), the length of each data member produced by the source code depends on the runtime environment, as shown in Table 5 on page 9.

Table 5. Comparison of data structure member lengths produced from the same code

Source:

```
struct li{
    long la;
    int ia;
} li;

struct lii{
    long la;
    int ia;
    int ib;
} lii;

struct ili{
    int ia;
    long la;
    int ib;
} ili;
```

ILP32 member lengths:

```
length li = 8    1
length lii = 12  3
length ili = 12  3
```

LP64 member lengths:

```
length li = 16   2
length lii = 16  3
length ili = 24  3
```

Notes:

1. **In a 32-bit environment, both `int` and `long int` have 4-byte alignments, so each of these members is aligned on 4-byte boundary. In accordance with the z/OS rule of alignment, the structure as a whole has a 4-byte alignment. The size of `struct li` is 8 bytes. See Figure 1 on page 10.**
2. **In a 64-bit environment, `int` has a 4-byte alignment and `long int` has an 8-byte alignment. In accordance with the z/OS rule of alignment, the structure as a whole has an 8-byte alignment. See Figure 1 on page 10.**
3. **The `struct lii` and the `struct ili` have the same members, but in a different member order. See Figure 2 on page 11 and Figure 3 on page 12. Because of the padding differences in each environment:**
   - **Under ILP32:**
     - **The size of `struct lii` is 12 bytes (4-byte long + 4-byte int + 4-byte int)**
     - **The size of `struct ili` is 12 bytes (4-byte int + 4-byte long + 4-byte int)**
   - **Under LP64:**
     - **The size of `struct lii` is 16 bytes (8-byte long + 4-byte int + 4-byte int)**
     - **The size of `struct ili` is 24 bytes (4-byte int + 4-byte pad + 8-byte long + 4-byte int + 4-byte pad)**

The ILP32 and LP64 alignments for the structs defined by the code shown in Table 5 on page 9 are compared in Figure 1 on page 10, Figure 2 on page 11, and Figure 3 on page 12.

Figure 1 on page 10 compares how `struct li` is aligned under ILP32 and LP64. The structure has two members:

- The first (member `la`) is of type `long`
- The second (member `ia`) is of type `int`

Under ILP32, each member is 4 bytes long and is aligned on a 4-byte boundary, making the structure 8 bytes long. Under LP64, member `la` is 8 bytes long and is aligned on an 8-byte boundary. Member `ia` is 4 bytes long, so the compiler inserts 4 padding bytes to ensure that the structure is aligned to the strictest alignment requirement for its largest member. Then, the structure can be used as part of an array under LP64.

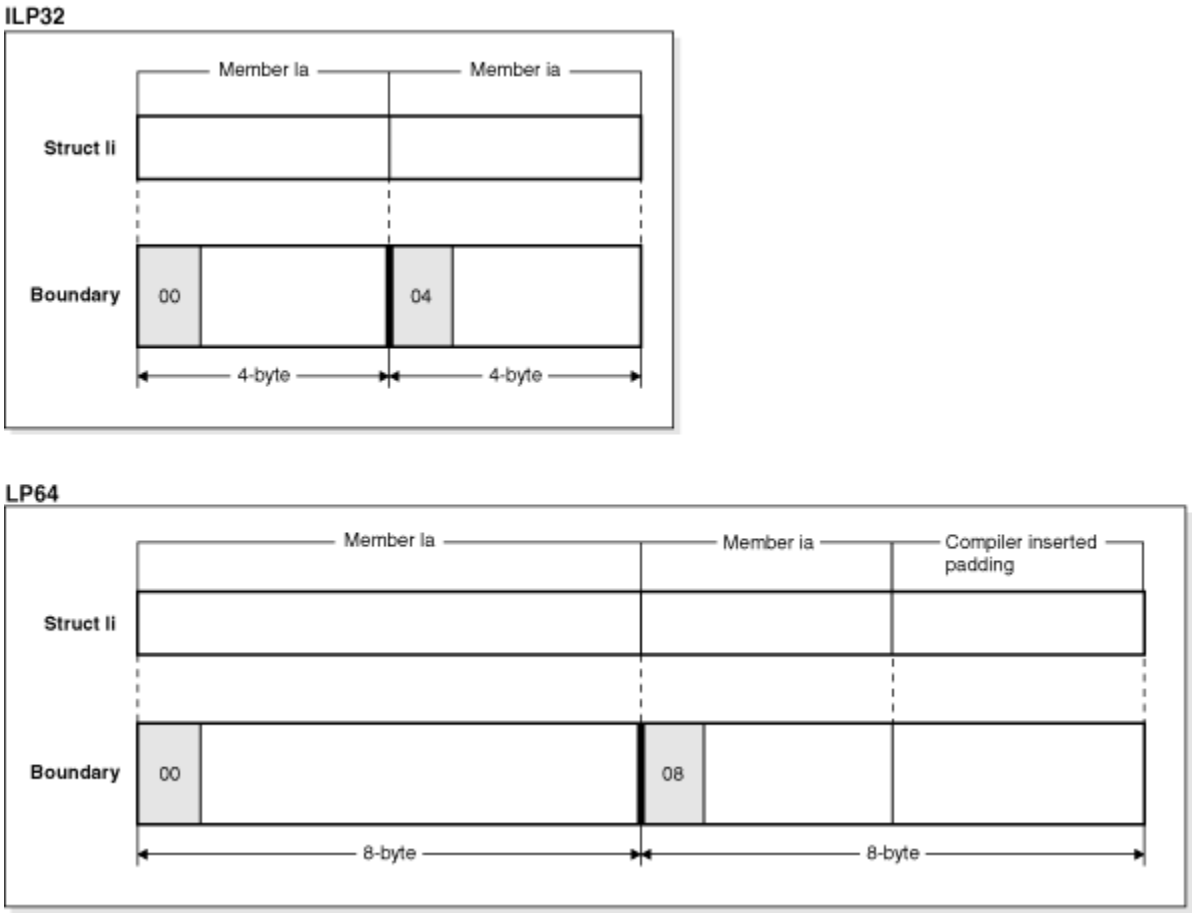


Figure 1. Comparison of struct li, alignments under ILP32 and LP64

Figure 2 on page 11 and Figure 3 on page 12 show structures that have the same members, but in a different order. Compare these figures to see how the order of the members impacts the size of the structures in each environment.

Figure 2 on page 11 compares how `struct lii` is aligned under ILP32 versus LP64. `struct lii` has three members:

- The first (member `la`) is of type `long`
- The second (member `ia`) and third (member `ib`) are of type `int`

Under ILP32, each member is 4 bytes long and is aligned on a 4-byte boundary, making the structure 12 bytes long. Under LP64, member la is 8 bytes long and is aligned on an 8-byte boundary. Member `ia` and member `ib` are each 4 bytes long, so the structure is 16 bytes long and can align on an 8-byte boundary without padding.

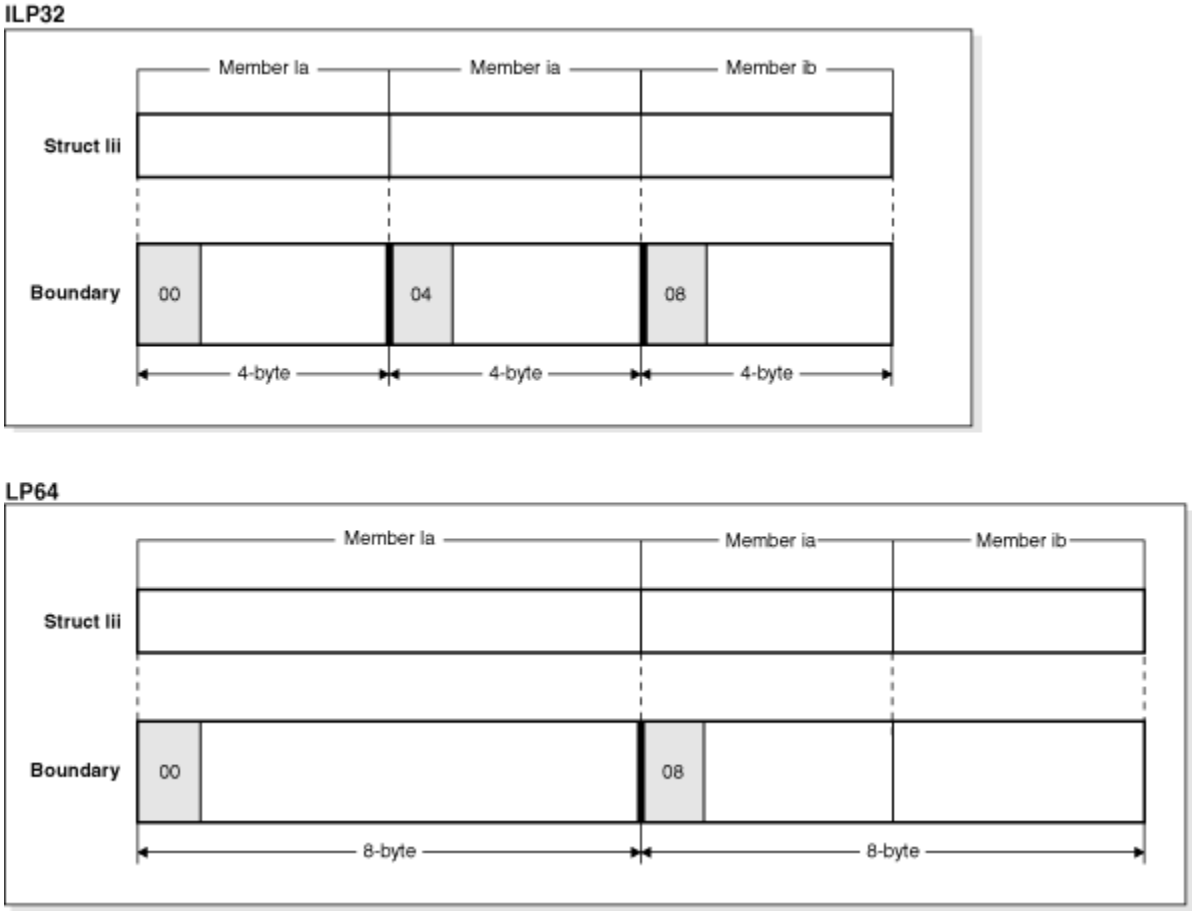


Figure 2. Comparison of struct lii alignments under ILP32 and LP64

Figure 3 on page 12 compares how `struct ili` is aligned under ILP32 and LP64. `struct ili` has three members:

- The first (member `ia`) is of type `int`
- The second (member `la`) is of type `long`
- The third (member `ib`) is of type `int`

Under ILP32, each member is 4 bytes long and is aligned on a 4-byte boundary, making the structure 12 bytes long. Under LP64, the compiler inserts padding after both member `ia` and member `ib`, so that each member with padding is 8 bytes long (member `la` is already 8 bytes long) and are aligned on 8-byte boundaries. The structure is 24 bytes long.

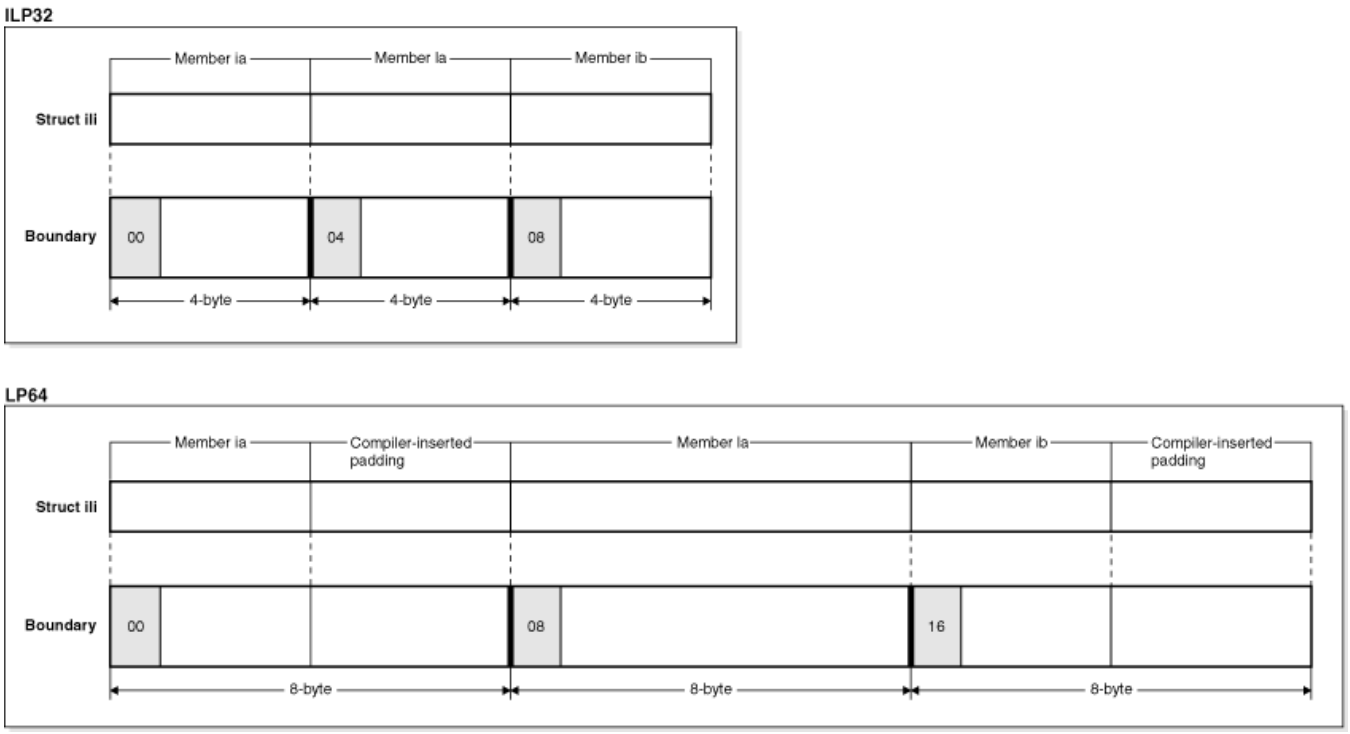


Figure 3. Comparison of struct ili alignments under ILP32 and LP64

Data type assignment differences under ILP32 and LP64

Under ILP32, `int`, `long`, and pointer types have the same size and can be freely assigned to one another.

Under LP64, all pointer types are 8 bytes in size. Assigning pointers to `int` types and back again can result in a invalid address, and passing pointers to a function that expects an `int` type will result in truncation. For example, the following statement show an incorrect assignment.

```
int i;
int *p;
i = (int)p;
```

Note: The problem is harder to detect when casts are used. Although there is no warning message, the problem still exists.

Avoid making any of the following assumptions:

- A pointer type or a `long` type can fit into an integer type.
- A type that is derived from a pointer type can fit into a type derived from an integer type.
- The number of bits in a `long` type object is assumed, especially when shifting bits or doing bitwise operations.
- An integer can be passed to an unprototyped `long` or pointer parameter.
- A function that is not a prototype can return a pointer or `long`.

Portability issues with data types long and int

Under LP64, `long` and `int` data types are not interchangeable. The `long` type (and types derived from it) is 64 bits in size.

You should consider all types related to the `long` and `unsigned long` types. For example, `size_t`, which is used in many subroutines, is defined under LP64 as `unsigned long`.

Because of the difference in size for `int` and `long` under LP64, conversions to `long` from other integral types might be executed differently that it was under ILP32.

Example of possible change of result after conversion from signed number to unsigned long

When a `signed char`, `signed short`, or `signed int` is converted to `unsigned long`, sign extension might result in a different unsigned value in 64-bit mode. The example in Table 6 on page 13 will yield 4294967295 (0xffffffff) under ILP32 but 18446744073709551615 (0xffffffffffffffff) under LP64, because of sign extension.

Table 6. Example of possible change of result after conversion from signed number to unsigned long

Source	`void myfunc(int i)` `{` `  unsigned long l = i;` `  return l;` `}` `void main()` `{` `  return myfunc(-1);` `}`
Compiler options	`metalc -Wc,"flag(i),warn64" -c warn2.c`
Output	`INFORMATIONAL CJT3743 ./warn2.c:3 64-bit portability:` ` possible change of result through conversion of int` ` type into unsigned long int type.`

Example of possible change of result after conversion from unsigned int variable to signed long

When an `unsigned int` variable with values greater than `INT_MAX` is converted to `signed long`, the results depend on whether the application is executed under ILP32 or under LP64. In the example in Table 7 on page 13:

- Under ILP32, the value `INT_MAX+1` will wrap around and yield -2147483648 (0x80000000)
- Under LP64, the value `INT_MAX+1` can be represented by an 8-byte `signed long` and will result in the correct value 2147483648 (0x80000000)

Table 7. Example of possible change of result after conversion from unsigned int variable to signed long

Source	`#include<limits.h>` `void myfunc(unsigned int i)` `{` `  long l = i;` `  return l;` `}` `void main()` `{` `  return myfunc(INT_MAX + 1);` `}`
Compiler options	`metalc -Wc,"flag(i),warn64" -c warn3.c`
Output	`INFORMATIONAL CJT3743 ./warn3.c:4 64-bit portability:` ` possible change of result through conversion of` ` unsigned int type into long int type.`

Example of possible change of result after conversion from signed long long variable to unsigned long

When a `signed long long` variable with values either greater than `UINT_MAX` or less than 0 is converted to `unsigned long`, truncation will not occur under LP64. The example in Table 8 on page 14 will yield:

- 4294967295 (0xffffffff) 0 (0x0) under ILP32
- 18446744073709551615 (0xffffffffffffffff) 4294967296 (0x100000000) under LP64

Table 8. Example of possible change of result after conversion from signed long long variable to unsigned long

Source	`#include<limits.h>` `void myfunc(signed long long ll)` `{` `  unsigned long l = ll;` `  return l;` `}` `void main()` `{` `  return myfunc(-1) + myfunc(UINT_MA X+ 1ll);` `}`
Compiler options	`metalc -Wc,"flag(i),warn64" -c warn4.c`
Output	`INFORMATIONAL CJT3743 ./warn4.c:4 64-bit portability:` ` possible change of result through conversion of long` ` long int type into unsigned long int type.`

Example of possible change of result after conversion from unsigned long long variable to unsigned long

Under LP64, when an `unsigned long long` variable with values greater than `UINT_MAX` is converted to `unsigned long`, truncation will not occur.

Table 9. Example of possible change of result after conversion from unsigned long long variable to unsigned long

Source	`#include<limits.h>` `void myfunc(unsigned long long ll)` `{` `  unsigned long l = ll;` `  return l;` `}` `void main()` `{` `  return myfunc(UINT_MAX + 1ull);` `}`
ILP32 output	Return value: `0 (0x0)` **Note:** The higher order word is truncated.
LP64 output	Return value: `4294967296 (0x100000000)` **Note:** There is no truncation.

Example of possible change of result after conversion from signed long long variable to signed long

Under LP64, when a `signed long long` variable with values less than `INT_MIN` or greater than `INT_MAX` is converted to `signed long`, truncation does not occur.

Table 10.

Source

```
#include<limits.h>
void myfunc(signed long long ll)
{
   signed long l = ll;

   return l;
}
```

```
void main()
{
   myfunc(INT_MIN - 1ll);
}
```

```
void main()
{
   myfunc(INT_MAX + 1ll);
}
```

Compiler options

```
metalc -Wc,"flag(i),warn64" -c warn5.c
```

ILP32 output

```
INFORMATIONAL CJT3743 ./warn5.c:4 64-bit portability:
 possible change of result through conversion of long
 long int type into long int type.
```

Return value: -2147483649 (0xffffffff7fffffff)	Return value: 2147483648 (0x80000000)

Note: The higher order word is truncated.

LP64 output

```
INFORMATIONAL CJT3743 ./warn5.c:4 64-bit portability:
 possible change of result through conversion of long
 long int type into long int type.
```

Return value: -2147483649 (0xffffffff7fffffff)	Return value: 2147483648 (0x80000000)

Note: There is no truncation.

Example of possible change of result after conversion from unsigned long long variable to signed long

Under LP64, when an `usigned long long` variable with values greater than `INT_MAX` is converted to `signed long`, truncation does not occur.

Table 11. Example of possible change of result after conversion from unsigned long long variable to signed long

Source

```
#include<limits.h>
void myfunc(unsigned long long ll)
{
    signed long l = ll;

   return l;
}
void main()
{
    return myfunc(INT_MAX + 1ull);
}
```

Compiler options

```
metalc -Wc,"flag(i),warn64" -c warn6.c
```

Table 11. Example of possible change of result after conversion from unsigned long long variable to signed long (continued)

ILP32 output	`INFORMATIONAL CJT3743 ./warn6.c:4 64-bit portability: possible change of result through conversion of unsigned long long int type into long int type.` Return value: `-2147483648 (0x80000000)` **Note:** The value `INT_MAX+1ull` will wrap around.
LP64 output	`INFORMATIONAL CJT3743 ./warn6.c:4 64-bit portability: possible change of result through conversion of unsigned long long int type into long int type.` Return value: `2147483648 (0x80000000)` **Note:** The value `INT_MAX+1ull` can be represented by an 8-byte `signed long` and will result in the correct value.

Pointer declarations when 32-bit and 64-bit applications share header files

In 64-bit data models, pointer sizes are always 64 bits. There is no standard language syntax for specifying mixed pointer size. However, it might be necessary to specify the size of a pointer type to help migrate a 32-bit application (for example, when libraries share a common header between 32-bit and 64-bit applications).

The Enterprise Metal C for z/OS compiler reserves two pointer size qualifiers:

- `__ptr32`
- `__ptr64`

The size qualifier `__ptr64` is not currently used; it is reserved so that a program cannot use it. The size qualifier `__ptr32` declares a pointer to be 32 bits in size. This is ignored under ILP32.

Examples of pointer declarations that can be made under LP64:

```
int * __ptr32 p; /* 32-bit pointer */ 1, 3
int * r;        /* 64-bit pointer, default to the model's size */ 4
int * __ptr32 const q; /* 32-bit const pointer */ 1, 2, 3
```

Notes:

1. The qualifier qualifies the * before it.
2. q is a 32-bit constant pointer to an integer.
3. When `__ptr32` is used, the program expects that the address of the pointer variable is less than or equal to 31 bits.
4. If a pointer declaration does not have the size qualifier, it defaults to the size of the data model.

Potential pointer corruption

When porting a program from ILP32 to LP64, be aware of the following potential problems:

- An invalid address might be the result of either of the following actions:
  - Assigning an integer (4 bytes) or a 4-byte hexadecimal constant to a pointer type variable (8 bytes)
  - Casting a pointer to an integer type

  Note: An invalid address causes errors when the pointer is dereferenced.
- If you compare an integer to a pointer, you might get unexpected results.
- Data truncation might result if you convert pointers to signed or unsigned integers with the expectation that the pointer value will be preserved.

- If return values of functions that return pointers are assigned to an integer type, those return values will be truncated.
- If code assumes that pointers and integers are the same size (in an arithmetic context), there will be problems. Pointer arithmetic is often a source of problems when migrating code. The ISO C standard dictates that incrementing a pointer adds the size of the data type to which it points to the pointer value. For example, if the variable p is a pointer to `long`, the operation `(p+1)` increments the value of p by 4 bytes (in 32-bit mode) or by 8 bytes (in 64-bit mode). Therefore, casts between `long*` and `int*` are problematic because of the size differences between pointer objects (32 bits versus 64 bits).

Potentially incorrect pointer-to-int and int-to-pointer conversions

Before porting code, It is important to test the ILP32 code to determine if any code paths would have incorrect results under LP64. For example:

- When a pointer is explicitly converted to an integer, truncation of the high-order word occurs.
- When an integer is explicitly converted to a pointer, the pointer might not be correct, which could result in invalid memory access when the pointer is dereferenced.

Potential truncation problem with a pointer cast conversion

As Table 12 on page 17 shows, truncation problems can occur when converting between 64-bit and 32-bit data objects. Because `int` and `long` are both 32 bits under ILP32, a mixed assignment or conversion between these data types did not represent any problem. However, under LP64, a mixed assignment or conversion does present problems because `long` is larger in size than `int`. Without an explicit cast, the compiler is unable to determine whether the narrowing of assignment is intended. If the value l is always within the range representable by an int, or if the truncation is intended by design, use an explicit cast to silent the `WARN64` message that you will receive for this code.

Table 12. Example of truncation problem with a pointer cast conversion

Source	`void myfunc(long l)` `{` `  int i = l;` `}`
Compiler options	`metalc -Wc,"flag(i),warn64" -c warn1.c`
Output	`WARNING CJT3742 ./warn1.c:3 64-bit portability:` `possible loss of digits through conversion of long` `int type into int type.`

Potential loss of data in constant expressions

A loss of data can occur in some constant expressions because of lack of precision. These types of problems are very hard to find and might be unnoticed. It is possible to write data-neutral code that can be compiled under both ILP32 and LP64.

When coding constant expressions, you must be very explicit about specifying types and use the constant suffixes (u, U, `l`, L, `ll`, LL) to specify types, as shown in Table 13 on page 18. You could also use casts to specify the type of a constant expression.

It is especially important to code constant expressions carefully when you are porting programs to a 64-bit environment because integer constants might have different types when compiled in 64-bit mode. The ISO C standard states that the type of an integer constant, depending on its format and suffix, is the first (that is, smallest) type in the corresponding list that will hold the value. The number of leading zeros does not influence the type selection. Table 13 on page 18 describes the type of an integer constant according to the ISO standards.

Table 13. Type of an integer constant		
Suffix	**Decimal constant**	**Octal or hexadecimal constant**
unsuffixed	`int` `long` `unsigned long`	`int` `unsigned int` `long` `unsigned long`
u or U	`unsigned int` `unsigned long`	`unsigned int` `unsigned long`
l or L	`long` `unsigned long`	`long` `unsigned long`
Both u or U and l or L	`unsigned long`	`unsigned long`
ll or LL	`long long`	`long long` `unsigned long long`
Both u or U and ll or LL	`unsigned long long`	`unsigned long long`

Note: Under LP64, a change in the type of a constant in an expression might cause unexpected results because `long` is equal to `long long`. For example, an unsuffixed hexadecimal constant that can be represented only by an `unsigned long` in 32-bit mode can fit within a `long` in 64-bit mode.

Data alignment problems when structures are shared

Modern processor designs usually require data in memory to be aligned to their natural boundaries, in order to gain the best possible performance. In most cases, the compiler ensures proper alignment by inserting padding bytes immediately in front of the misaligned data. Although the padding bytes do not affect the integrity of the data, they might result in an unexpected layout, which affects the size of structures and unions.

Because both pointer size and long size are doubled in 64-bit mode, structures and unions containing them as members are larger than they are in 32-bit mode.

Figure 4 on page 19 illustrates how the compiler treats the source code shown in #unique_45/unique_45_Connect_42_codeshareptrs under ILP32 and LP64. Because the pointer is a different size in each environment, they are aligned on different boundaries. This means that if the code is compiled under both ILP32 and LP64, there are likely to be alignment problems. Figure 5 on page 22 illustrates the solution, which is to define pad members of type character that prevent the possibility of data misalignment. Table 15 on page 21 shows the necessary modifications to the code in #unique_45/unique_45_Connect_42_codeshareptrs.

If the structure in #unique_45/unique_45_Connect_42_codeshareptrs is shared or exchanged among 32-bit and 64-bit processes, the data fields (and padding) of one environment will not match the expectations of the other, as shown in Figure 4 on page 19.

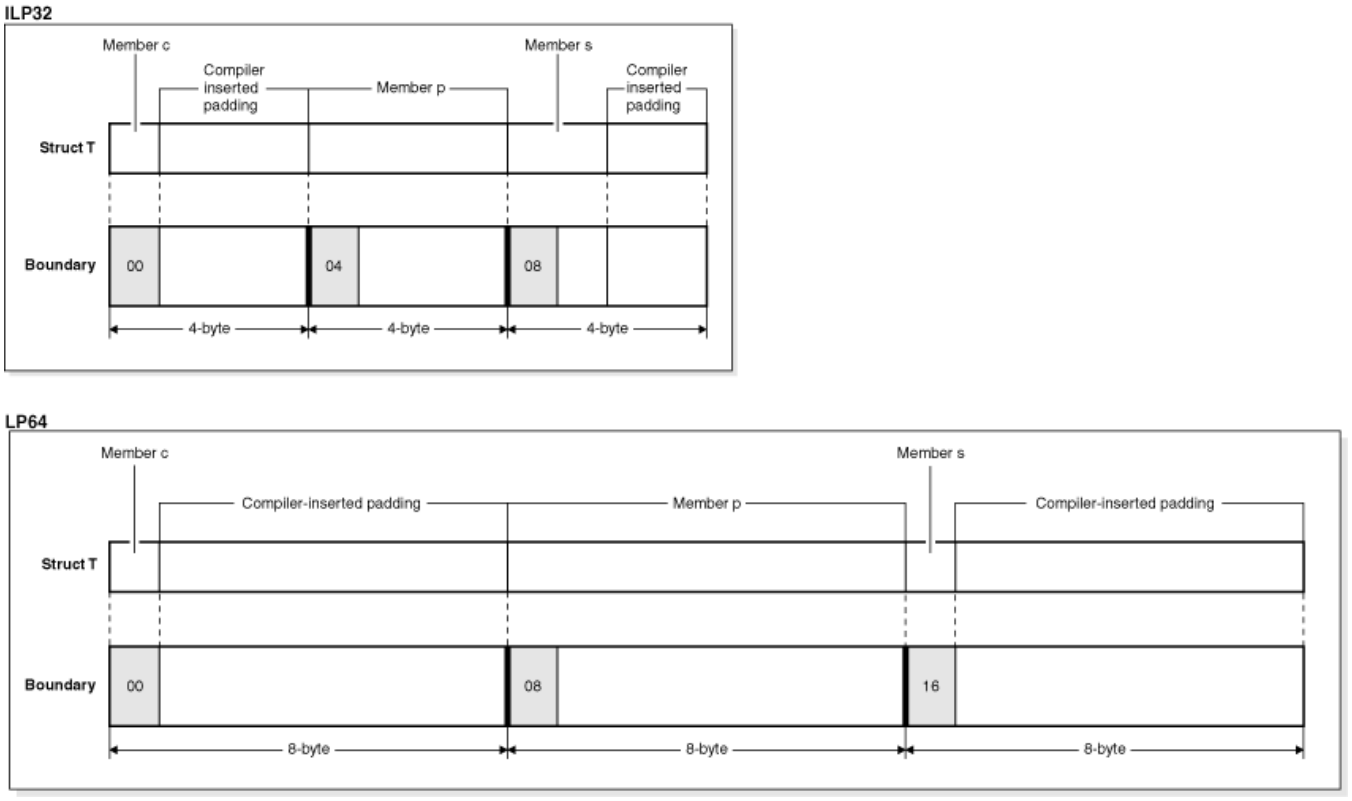


Figure 4. Example of potential alignment problems when a struct is shared or exchanged among 32-bit and 64-bit processes

Portability issues with unsuffixed numbers

When porting code, be aware that:

- Unsuffixed constants are more likely to become 8 bytes long if they are in hexadecimal.
- All constants that can impact any constant assignment must be explicitly suffixed.

Example of unexpected behavior resulting from use of unsuffixed numbers

This causes some operations, such as one that compares `sizeof(4294967295)` to another value, to return 8. If you add the suffix U to the number (4294967295U), the compiler can parse it as `unsigned int`.

Table 14. Example of unexpected behavior resulting from use of unsuffixed numbers

Source	`#include <limits.h>` `void main(void) {` `  long l = LONG_MAX;` `}`
Under ILP32	`size(2147483647) = 4` `size(2147483648) = 4` `size(4294967295U) = 4` `size(-1) = 4` `size(-1L) = 4` `LONG_MAX = 2147483647`

Table 14. Example of unexpected behavior resulting from use of unsuffixed numbers (continued)	
Under LP64	`size(2147483647) = 4` `size(2147483648) = 8` `size(4294967295U) = 4` `size(-1) = 4` `size(-1L) = 8` `LONG_MAX = -1`

Example of how a suffix causes the compiler to parse the number differently under ILP32 than under LP64

Example: A number like `4294967295 (UINT_MAX)`, when parsed by the compiler, will be

- An `unsigned long` under ILP32
- A `signed long` under LP64

Using a LONG_MAX macro in a sprintf subroutine

The `sprintf` subroutine format string for a 64-bit integer is different than the string used for a 32-bit integer. Programs that do these conversions must use the proper format specifier.

Under LP64, you must also consider the maximum number of digits of the `long` and `unsigned long` types. The `ULONG_MAX` is twenty digits long, and the `LONG_MAX` is nineteen digits.

Programming for portability between ILP32 and LP64

When you want to program for portability between the ILP32 and LP64 environments, you can use the following strategies:

- Header files to provide type definitions
- Suffixes and explicit types to prevent unexpected behavior
- Defining pad members to avoid data alignment problems
- Prototypes to avoid debugging problems
- Conditional compiler directive for preprocessor macro selection

Using header files to provide type definitions

The header file `inttypes.h` provides type definitions for integer types that are guaranteed to have a specific size (for example, `int32_t` and `int64_t`, and their unsigned variations). Consider using those type definitions if your program code relies on types with specific sizes.

There are many ways to use headers to handle code that is portable between ILP32 and LP64. You can minimize the amount of conditional compilation code and avoid having totally different sections of code for a ILP32 and LP64 structure definitions if you adopt a coding convention that suits your environment.

If you provide a library to your application users and ship header files that define the application programming interface of the library, consider shipping a single set of headers that can support both 32-bit and 64-bit versions of your library. You can use the type definitions in `inttypes.h`. For example, if you are currently shipping 32-bit versions of your header files, you could:

- Replace all fields of type `long` with type `int32_t` (or another 32-bit type)
- Similarly replace all fields for the unsigned variation
- If you cannot let a 64-bit application use a 64-bit pointer for a field, use the `__ptr32` qualifier.

Using suffixes and explicit types to prevent unexpected behavior

The C language limit (in `limits.h`) is different under LP64 than it is under ILP32. As the following example shows, you can prevent unexpected behavior by an application by using suffixes and explicit types with all numbers.

```
#ifdef _LP64
#define LONG_MAX (9223372036854775807L)
#define LONG_MIN (-LONG_MAX - 1)
#define ULONG_MAX (18446744073709551615U)
#else
#define LONG_MAX INT_MAX
#define LONG_MIN INT_MIN
#define ULONG_MAX (UINT_MAX)
#endif /* _LP64 */
```

Note: The output for `LONG_MAX` is not really -1. The reason for the -1 is that:

- The `sprintf` subroutine handles it as an integer
- (`LONG_MAX == (int)LONG_MAX`) returns a negative value

Defining pad members to avoid data alignment problems

If you want to allow the structure to be shared, you might be able to reorder the fields in the data structure to get the alignments in both 32-bit and 64-bit environments to match (as shown in Table 5 on page 9), depending on the data types used in the structure and the way in which the structure as a whole is used (for example, whether the structure is used as a member of another structure or as an array).

If you are unable to reorder the members of a structure, or if reordering alone cannot provide correct alignment, you can define paddings that force the members of the structure to fall on their natural boundaries regardless of whether it is compiled under ILP32 or LP64. A conditional compilation section is required whenever a structure uses data types that have different sizes in 32-bit and 64-bit environments.

The example in Table 15 on page 21 shows how the source code in #unique_45/ unique_45_Connect_42_codeshareptrs can be modified to avoid the data alignment problem.

Table 15. Example of source code that successfully shares pointers between ILP32 and LP64 programs

Source:	`struct T {` `  char c;` `  short s;` `  #if !defined(_LP64)` `  char pad1[4];` `  #endif` `  int *p;` `  #if !defined(_LP64)` `  char pad2[4];` `  #endif` `} t`
ILP32/ LP64 size and member layout:	`sizeof(t) = 16` `offsetof(t, c) = 0 sizeof(c) = 1` `offsetof(t, s) = 2 sizeof(s) = 2` `offsetof(t, p) = 8 sizeof(p) = 4`

Figure 5 on page 22 shows the member layout of the structure with user-defined padding. Because the pointer is a different size in each environment, it is aligned on different a boundary in each environment. This means that if the code is compiled under both ILP32 and LP64, there are likely to be alignment problems. This figure illustrates the solution, which is to define pad members of type character that prevent the possibility of data misalignment.

Note: When inserting paddings into structures, use an array of characters. The natural alignment of a character is 1-byte, which means that it can reside anywhere in memory.

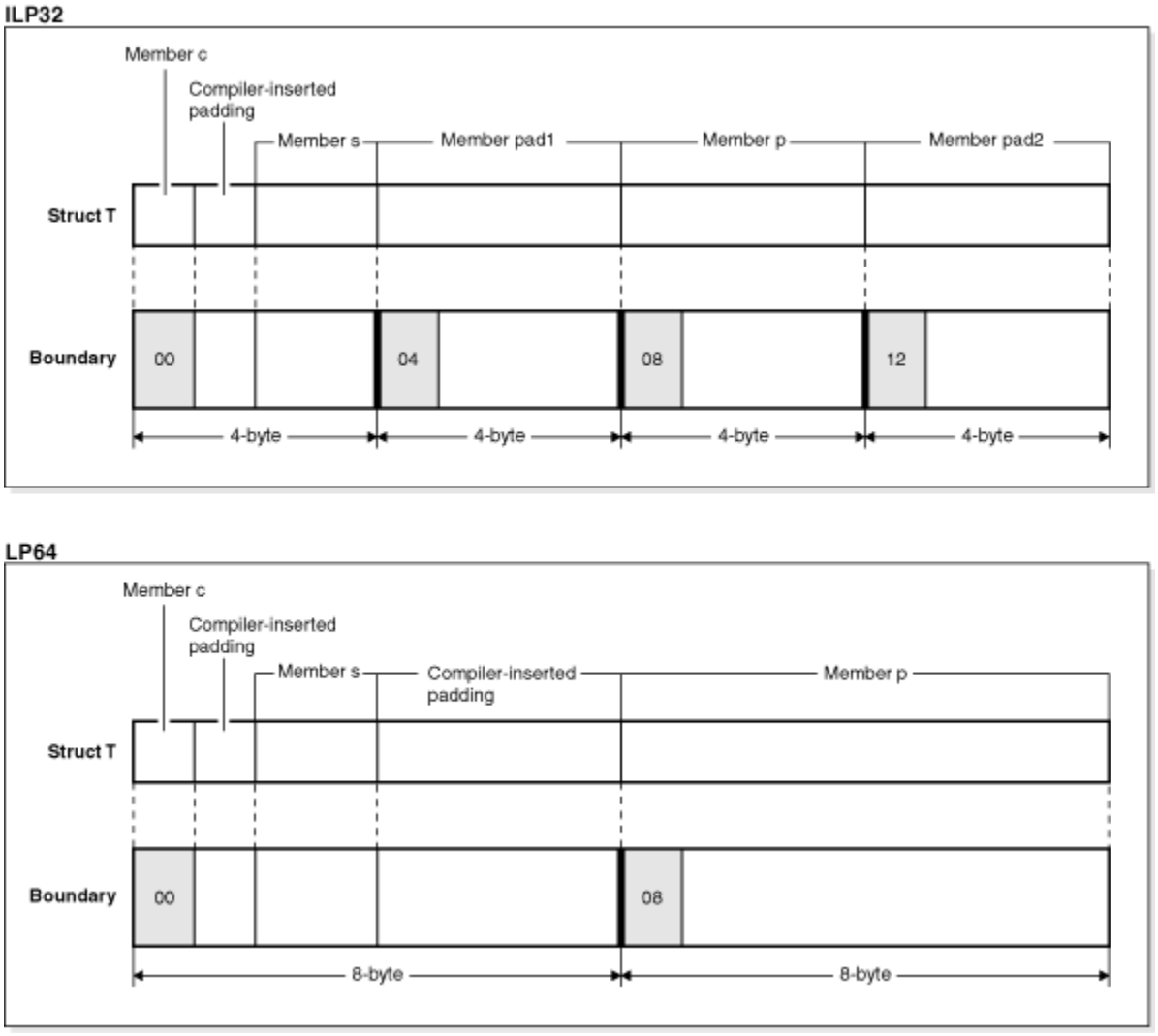


Figure 5. Example of user-defined data padding for a structure that is shared or exchanged among 32-bit and 64-bit processes

Using prototypes to avoid debugging problems

You can avoid complex debugging problems by ensuring that all functions are prototyped.

The C language provides a default prototype. If a function is not prototyped, it defaults to a function which returns an integer and has no information about the parameters.

A problem is that the default return type of `int` might not remain the same size as an associated pointer. For example, the function `malloc()` can cause truncation when an unprototyped function returns a pointer. This is because an unprototyped function is assumed to return an `int` (4 bytes).

Using a conditional compiler directive for preprocessor macro selection

When the compiler is invoked with the LP64 option, the preprocessor macro _LP64 is defined. When the compiler is invoked with the `ILP32` option, the macro _ILP32 is defined.

You can use a conditional compiler directive such as `#if defined _LP64` or `#ifdef _LP64` to select lines of code that are appropriate for the data model that is invoked.

Chapter 2. Reentrancy in Enterprise Metal C for z/OS

This information describes the concept of reentrancy. It tells you how to use reentrancy in C programs to help make your programs more efficient.

Reentrant programs are structured to allow multiple users to share a single copy of an executable module or to use an executable module repeatedly without reloading. C achieves reentrancy by splitting your program into two parts, which are maintained in separate areas of memory until the program terminates:

- The first part, which consists of executable code and constant data, does not change during program execution.
- The second part contains persistent data that can be altered. This part includes the dynamic storage area (DSA) and a piece of storage known as the writable static area.

If the program is installed in the Link Pack Area (LPA) or Extended Link Pack Area (ELPA) of your operating system, only a single copy of the first (constant or reentrant) part exists within a single address space. This occurs regardless of the number of users that are running the program simultaneously. This reentrant part may be shared across address spaces or across sessions. In this case, the executable module is loaded only once. Separate concurrent invocations of the program share or reenter the same copy of the write-protected executable module. If the program is not installed in the LPA or ELPA area, each invocation receives a private copy of the code part, but this copy may not be write-protected.

The modifiable writable static part of the program contains:

- All program variables with the `static` storage class
- All program variables receiving the `extern` storage class
- All writable strings
- All variable pointers for imported variables (non-XPLINK)
- All function pointers for imported functions (RENT)
- All variable linkage descriptors to reference imported variables (non-XPLINK)

Each user running the program receives a private copy of the second (data or non-reentrant) part. This part, the data area, is modifiable by each user.

The code part of the program contains:

- Executable instructions
- Read-only constants
- Global objects compiled with the `#pragma variable(identifier, NORENT)`

  Note: The ROCONST compiler option implicitly inserts a `#pragma variable(identifier, NORENT)` for `const` qualified variables.

Natural or constructed reentrancy

Natural reentrancy
C programs that contain no references to the writable static objects listed in a previous section have natural reentrancy. You do not need to compile naturally reentrant C programs with the RENT compiler option or bind them with the binder.

Constructed reentrancy
C programs that contain references to writable static objects, can have constructed reentrancy. You must bind these programs with the binder. You must use the RENT compiler option.

Limitations of constructed reentrancy for C programs

Even if a C program is large and will have more than one user at the same time, there are also these limitations to consider:

- If the binder is used, the resultant program must reside in a PDSE or UNIX file system. If a PDSE member should be installed into LPA or ELPA, it can only be installed into dynamic LPA.
- A system programmer can install only the shared portion of your program in the LPA or ELPA of your operating system.

Controlling external static in C programs

Certain program variables with the `extern` storage class may be constant and never written. If this is the case, every user does not need to have a separate copy of these variables. In addition, there may be a need to share constant program variables between C and another language.

You can force an external variable to be the part of the program that includes executable code and constant data by using the `#pragma variable(varname, NORENT)` directive. The program fragment in Figure 6 on page 24 illustrates how this is accomplished.

```
#pragma options(RENT)

#pragma variable(rates, NORENT)
extern float rates[5] = { 3.2, 83.3, 13.4, 3.6, 5.0 };

extern float totals[5];

int main(void) {
    /* ... */
}
```

Figure 6. Controlling external static

In this example, the source file is compiled with the RENT option. The external variable rates are included in the executable code because `#pragma variable(rates, NORENT)` is specified. The variable totals are included with the writable static. Each user has a copy of the array totals, and the array rates are shared among all users of the program.

The `#pragma variable(varname, NORENT)` does not apply to, and has no effect on, program variables with the `static` storage class. Program variables with the `static` storage class are always included in the writable static.

When specifying `#pragma variable(varname, NORENT)`, ensure that this variable is never written; if it is written, program exceptions or unpredictable program behavior may result. In addition, you must include `#pragma variable(varname, NORENT)` in every source file where the variable is referenced or defined. It is good practice to put these pragmas in a common header file.

Note: You can also use the keyword `const` to ensure that a variable is not written. See the `const` type qualifier in for more information.

The ROCONST compiler option has the same effect as specifying the `#pragma variable (var_name, NORENT)` for all constant variables (i.e. `const` qualified variables). The option gives the compiler the choice of allocating `const` variables outside of the Writable Static Area (WSA). For more information, see ROCONST | NOROCONST in .

Controlling writable strings

In a large number of C programs, character strings may be constant and never written to. If this is the case, every user does not need a separate copy of these strings.

You can force all strings in a given source file to be the part of the program that includes executable code and constant data by using `#pragma strings(readonly)` or the `ROSTRING` compiler option. #unique_59/unique_59_Connect_42_makecon illustrates one way to make the strings constant.

Ensure that you do not write to read-only strings. The following code tries to overwrite the literal string `abcd` because `chrs` is just a pointer:

```
char chrs[]= "abcd";
memcpy(chrs,"ABCD",4);
```

Program exceptions or unpredictable program behavior may result if you attempt to write to a string constant.

The `ROSTRING` compiler option has the same effect as `#pragma strings(readonly)` in the program source. For more information, see ROSTRING | NOROSTRING in .

Chapter 3. Using vector programming support

Enterprise Metal C for z/OS supports vector programming by making use of the Vector Facility for z/Architecture®.

The compiler supports vector processing technologies through language extensions, based on the AltiVec Programming Interface specification and OpenPower ABI Vector Programming Interface specification with suitable changes and extensions.

This chapter describes Enterprise Metal C for z/OS language extensions for vector processing support, including compiler options, vector data types and operators, macro, and built-in functions.

Options

The vector language extensions are enabled only when all of the following options are in effect:

- ARCH(11) or a higher level
- FLOAT(AFP(NOVOLATILE))
- VECTOR

Notes:

- The VECTOR option implies LANGLVL(LONGLONG), which enables the `vector bool long long`, `vector signed long long`, `vector unsigned long long`, `__vector bool long long`, `__vector signed long long`, and `__vector unsigned long long` data types.
- The `vector float`, `__vector float`, `vector double`, and `__vector double` data types are only available with FLOAT(IEEE).
- The `vector float` and `__vector float` data types are available at a minimum ARCH level of 12.

For more information about these compiler options, see .

Macro

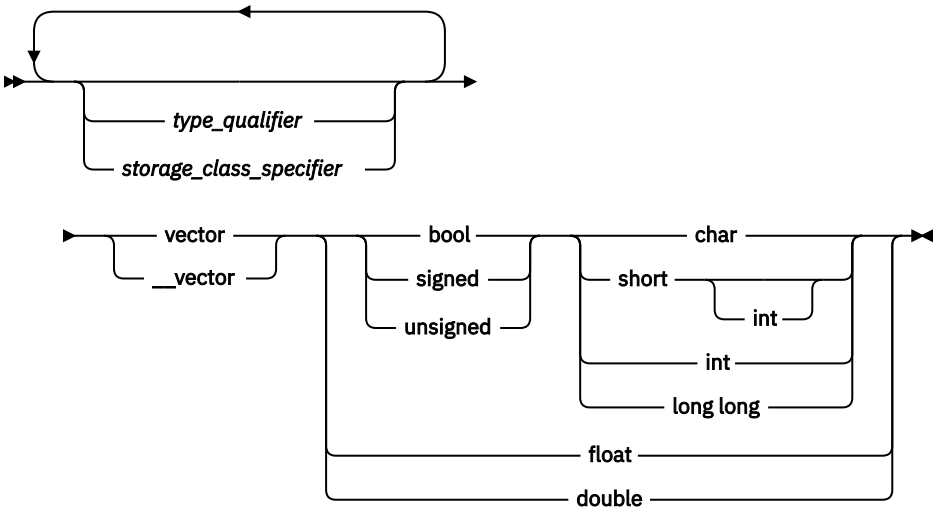
The __VEC__ macro indicates the support for vector data types. The predefined value of __VEC__ is 10402.

Vector data types

Vector programming is supported to provide an efficient and expressive mechanism for programmers to make use of the Vector Facility for z/Architecture from the C programming languages. This section describes the supported vector data types.

In this syntax, type qualifiers and storage class specifiers can precede the keyword `vector` (or its alternative spelling, `__vector`) in a declaration. Most of the legal forms of the syntax are captured in the following diagram. Some variations have been omitted from the diagram for the sake of clarity: type qualifiers such as `const` and storage class specifiers such as `static` can appear in any order within the declaration, as long as neither immediately follows the keyword `vector` (or `__vector`).

Vector declaration syntax



Notes:

- Both the keywords `vector` and `__vector` are recognized with the option VECTOR(TYPE); while only `__vector` is recognized with the option VECTOR(NOTYPE).
- The keyword `bool` is recognized as a valid type specifier only when preceded by the keyword `vector` or `__vector`.
- Duplicate type specifiers are ignored in a vector declaration context.
- The `vector bool long long`, `vector signed long long`, `vector unsigned long long`, `__vector bool long long`, `__vector signed long long`, and `__vector unsigned long long` data types are only available with LANGLVL(LONGLONG), which is implied by the VECTOR option.
- The vector floating-point types, which include `vector float`, `__vector float`, `vector double`, and `__vector double`, are available only with FLOAT(IEEE).
- The `vector float` and `__vector float` data types are available at a minimum ARCH level of 12.

The following table lists the supported vector data types, the size and possible values for each type.

Table 16. Vector data types

Type	Interpretation of content	Range of values
vector unsigned char	16 unsigned char	0..255
vector signed char	16 signed char	-128..127
vector bool char	16 unsigned char	0 (FALSE), 255 (TRUE)
vector unsigned short	8 unsigned short	0..65535
vector unsigned short int		
vector signed short	8 signed short	-32768..32767
vector signed short int		
vector bool short	8 unsigned short	0 (FALSE), 65535 (TRUE)
vector bool short int		
vector unsigned int	4 unsigned int	$0..2^{32}-1$
vector signed int	4 signed int	$-2^{31}..2^{31}-1$
vector bool int	4 unsigned int	0 (FALSE), $2^{32}-1$ (TRUE)

Table 16. Vector data types (continued)		
Type	**Interpretation of content**	**Range of values**
vector unsigned long long	2 unsigned long long	$0..2^{64}-1$
vector signed long long	2 signed long long	$-2^{63}..2^{63}-1$
vector bool long long	2 unsigned long long	0 (FALSE), $2^{64}-1$ (TRUE)
vector float	4 float	32-bit IEEE-754 single precision binary floating-point values
vector double	2 double	64-bit IEEE-754 double precision binary floating-point values

All vector types are aligned on an 8-byte boundary. An aggregate that contains one or more vector types is aligned on an 8-byte boundary, and padded, if necessary, so that each member of vector type is also 8-byte aligned.

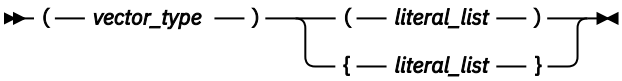
Language extensions

The C language is extended to support expressions and operations that are required to act on vector data types.

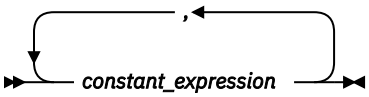
Vector literals

A vector literal is a constant expression for which the value is interpreted as a vector type. The data type of a vector literal is represented by a parenthesized vector type, and its value is a set of constant expressions that represent the vector elements and are enclosed in parentheses or braces. When all vector elements have the same value, the value of the literal can be represented by a single constant expression. You can initialize vector types with vector literals.

Vector literal syntax



literal_list



The *vector_type* is a supported vector type; see “Vector data types” on page 27 for a list of these.

The *literal_list* can be either of the following expressions:

- A single expression.

  If the single expression is enclosed with parentheses, all elements of the vector are initialized to the specified value. If the single expression is enclosed with braces, the first element of the vector is initialized to the specified value, and the remaining elements of the vector are initialized to 0.
- A comma-separated list of expressions. Each element of the vector is initialized to the respectively specified value.

  The number of constant expressions is determined by the type of the vector and whether it is enclosed with braces or parentheses.

  If the comma-separated list of expressions is enclosed with braces, the number of constant expressions can be equal to or less than the number of elements in the vector. If the number of constant expressions is less than the number of elements in the vector, the values of the unspecified elements are 0.

If the comma-separated list of expressions is enclosed with parentheses, the number of constant expressions must match the number of elements in the vector as follows:

2
For `vector unsigned long long`, `vector signed long long`, `vector bool long long`, and `vector double` types.

4
For `vector unsigned int`, `vector signed int`, `vector bool int`, and `vector float` types.

8
For `vector unsigned short`, `vector signed short`, and `vector bool short` types.

16
For `vector unsigned char`, `vector signed char`, and `vector bool char` types.

The following table shows the supported vector literals and how the compiler interprets them to determine their values.

Table 17. Vector literals

Syntax	**Interpreted by the compiler as**
(vector unsigned char)(*unsigned int*)	A list of 16 unsigned 8-bit quantities that all have the value of the single integer.
(vector unsigned char)(*unsigned int*, ...) (vector unsigned char){*unsigned int*, ...}	A list of 16 unsigned 8-bit quantities with the value specified by each of the 16 integers.
(vector signed char)(*int*)	A list of 16 signed 8-bit quantities that all have the value of the single integer.
(vector signed char)(*int*, ...) (vector signed char){*int*, ...}	A list of 16 signed 8-bit quantities with the value specified by each of the 16 integers.
(vector bool char)(*unsigned int*)	A list of 16 unsigned 8-bit quantities that all have the value of the single integer.
(vector bool char)(*unsigned int*, ...) (vector bool char){*unsigned int*, ...}	A list of 16 unsigned 8-bit quantities with a value specified by each of 16 integers.
(vector unsigned short)(*unsigned int*)	A list of 8 unsigned 16-bit quantities that all have the value of the single integer.
(vector unsigned short)(*unsigned int*, ...) (vector unsigned short){*unsigned int*, ...}	A list of 8 unsigned 16-bit quantities with a value specified by each of the 8 integers.
(vector signed short)(*int*)	A list of 8 signed 16-bit quantities that all have the value of the single integer.
(vector signed short)(*int*, ...) (vector signed short){*int*, ...}	A list of 8 signed 16-bit quantities with a value specified by each of the 8 integers.
(vector bool short)(*unsigned int*)	A list of 8 unsigned 16-bit quantities that all have the value of the single integer.
(vector bool short)(*unsigned int*, ...) (vector bool short){*unsigned int*, ...}	A list of 8 unsigned 16-bit quantities with a value specified by each of the 8 integers.

Table 17. Vector literals (continued)	
Syntax	**Interpreted by the compiler as**
(vector unsigned int)(*unsigned int*)	A list of 4 unsigned 32-bit quantities that all have the value of the single integer.
(vector unsigned int)(*unsigned int*, ...) (vector unsigned int){*unsigned int*, ...}	A list of 4 unsigned 32-bit quantities with a value specified by each of the 4 integers.
(vector signed int)(*int*)	A list of 4 signed 32-bit quantities that all have the value of the single integer.
(vector signed int)(*int*, ...) (vector signed int){*int*, ...}	A list of 4 signed 32-bit quantities with a value specified by each of the 4 integers.
(vector bool int)(*unsigned int*)	A list of 4 unsigned 32-bit quantities that all have the value of the single integer.
(vector bool int)(*unsigned int*, ...) (vector bool int){*unsigned int*, ...}	A list of 4 unsigned 32-bit quantities with a value specified by each of the 4 integers.
(vector unsigned long long)(*unsigned long long*)	A list of 2 unsigned 64-bit quantities that both have the value of the single long long.
(vector unsigned long long)(*unsigned long long*, ...) (vector unsigned long long){*unsigned long long*, ...}	A list of 2 unsigned 64-bit quantities specified with a value by each of the 2 unsigned long longs.
(vector signed long long)(*signed long long*)	A list of 2 signed 64-bit quantities that both have the value of the single long long.
(vector signed long long)(*signed long long*, ...) (vector signed long long){*signed long long*, ...}	A list of 2 signed 64-bit quantities with a value specified by each of the 2 long longs.
(vector bool long long)(*unsigned long long*)	A list of 2 boolean 64-bit quantities with a value specified by the single unsigned long long.
(vector bool long long)(*unsigned long long*, ...) (vector bool long long){*unsigned long long*, ...}	A list of 2 boolean 64-bit quantities with a value specified by each of the 2 unsigned long longs.
(vector float)(*float*)	A set of 4 32-bit IEEE-754 single-precision binary floating-point quantities that all have the value of the single float.
(vector float)(*float*, ...) (vector float){*float*, ...}	A set of 4 32-bit IEEE-754 single-precision binary floating-point quantities with a value specified by each of the 4 floats.
(vector double)(*double*)	A list of 2 64-bit IEEE-754 double-precision binary floating-point quantities that both have the value of the single double.
(vector double)(*double*, *double*) (vector double){*double*, *double*}	A list of 2 64-bit IEEE-754 double-precision binary floating-point quantities with a value specified by each of the 2 doubles.

Note: The value of an element in a vector bool is FALSE if each bit of the element is set to 0 and TRUE if each bit of the element is set to 1.

For example, for an unsigned integer vector type, the literal could be either of the following:

```
(vector unsigned int)(10)   /* initializes all four elements to a value of 10  */
(vector unsigned int)(14, 82, 73, 700)    /* initializes the first element
                                             to 14, the second element to 82,
                                             the third element to 73, and the
                                             fourth element to 700   */
```

You can cast vector literals with the Cast operator `()`. Enclosing the vector literal to be cast in parentheses can improve the readability of the code. For example, you can use the following code to cast a `vector signed int` literal to a `vector unsigned char` literal:

```
(vector unsigned char)((vector signed int)(-1, -1, 0, 0))
```

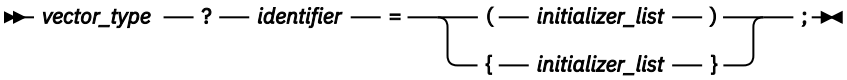
Initialization of vectors

A vector type can be initialized by a vector literal, or any expression having the same vector type. For example:

```
vector unsigned int v1;
vector unsigned int v2 = (vector unsigned int)(10);
v1 = v2;
```

A vector type can also be initialized by an initializer list.

Vector initializer list syntax



An initializer list enclosed with parentheses must have the same number of value as the number of elements of the vector type. The number of values in a braced initializer list must be less than or equal to the number of elements of the vector type. Any uninitialized element will be initialized to zero.

The following examples show vector initialization using initializer lists:

```
vector unsigned int v1 = {1}; // initialize the first element (4 bytes) of v1
                              // with 1 and the remaining 3 elments (12 bytes)
                              //     with zeros

vector unsigned int v2 = {1,2}; // initialize the first element (4 bytes) of v2
                                // with 1, the next element (4 bytes) with 2,
                                // and the remaining elements (8 bytes) with zeros

vector unsigned int v3 = {1,2,3,4}; // equivalent to the vector literal
                                    // (vector unsigned int) (1,2,3,4)
```

Unlike vector literals, the values in the initializer list do not have to be constant expressions unless the initialized vector variable has static duration. Thus, the following code is valid:

```
int i=1;
int function() { return 2; }
int main()
{
   vector unsigned int v1 = {i, function()};
   return 0;
}
```

typedef definitions for vector types

With a `typedef` declaration, you can define your own identifiers that can be used in place of type specifiers.

`typedef` definitions are extended to handle vector types, provided that vector support is enabled. A vector type can be used in a `typedef` definition, and the new type name can be used in the usual ways, except for declaring other vectors. In a vector declaration context, a `typedef` name is disallowed as a type specifier.

The following example illustrates a typical usage of `typedef` with vector types:

```
typedef vector unsigned short vushort4;
vushort4 v1;
```

Pointers

If you dereference a pointer to a vector data type, the standard behavior of either a load or a copy of the corresponding type is performed.

Pointer arithmetic can be used on vector data types. The result of the operation p+1 is a pointer to the next vector after the vector pointed to by p.

Unary expressions

Some unary expressions are extended for the vector data types.

Unary operators ++ -- + - ~

Vector data types can use some of the unary operators that are used with primitive data types, as outlined in the table below. These operators are not supported at global scope or for objects with static duration, and there is no constant folding. Each element in the vector has the operation applied to it.

Table 18. Unary operators

Operator	Integer vector types	Floating-point vector types	Bool vector types
++	Yes	Yes	No
−−	Yes	Yes	No
+	Yes	Yes	No
−	Yes 1	Yes 2	No
~	Yes	No	Yes

Notes:

1. Unary minus operator - treats unsigned integer vectors as signed integer vectors implicitly.
2. Unary minus operator - on floating-point vector types will not cause IEEE exception.

Related information

“Vector data types” on page 27

Address operator &

The & (address) operator can be used on the vector data types. It yields a pointer to the corresponding vector data type.

The __alignof__ operator

The `__alignof__` operator is a language extension to C99 that returns the position to which its operand is aligned.

The operand of `__alignof__` can be a vector type, provided that vector support is enabled. For example,

```
vector unsigned int v1 = (vector unsigned int)(10);
vector unsigned int *pv1 = &v1;
__alignof__(v1);      // vector type alignment: 8.
__alignof__(&v1);    // address of vector alignment: 4 (with ILP32) or 8 (with LP64).
__alignof__(*pv1);  // dereferenced pointer to vector alignment: 8.
__alignof__(pv1);    // pointer to vector alignment: 4 (with ILP32) or 8 (with LP64)
__alignof__(vector signed char);     // vector type alignment: 8.
```

When `__attribute__((aligned))` is used to increase the alignment of a variable of vector type, the value that is returned by the `__alignof__` operator is the alignment factor that is specified by `__attribute__((aligned))`.

The sizeof operator

The `sizeof` operator yields the size in bytes of the operand.

The operand of the `sizeof` operator can be a vector variable, a vector type, or the result of dereferencing a pointer to vector type, provided that vector support is enabled. In these cases, the return value of `sizeof` is always 16. For example,

```
vector bool int v1;
vector bool int *pv1 = &v1;
sizeof(v1);          // vector type: 16.
sizeof(&v1);       // address of vector: 4 (with ILP32) or 8 (with LP64).
sizeof(*pv1);     // dereferenced pointer to vector: 16.
sizeof(pv1);       // pointer to vector: 4 (with ILP32) or 8 (with LP64).
sizeof(vector double);     // vector type: 16.
```

The typeof operator

The `typeof` operator returns the type of its argument, which can be an expression or a type.

It is extended to accept a vector type as its operand, when vector support is enabled.

The vec_step operator

The `vec_step` operator takes a vector type operand and returns an integer value representing the amount by which a pointer to a vector element should be incremented in order to move by 16 bytes (the size of a vector), or equivalently, the number of elements in the vector. The following table provides a summary of values by data type.

Table 19. Increment value for vec_step by data type

vec_step expresssion	**Value**
`vec_step(vector unsigned char)` `vec_step(vector signed char)` `vec_step(vector bool char)`	16
`vec_step(vector unsigned short)` `vec_step(vector signed short)` `vec_step(vector bool short)`	8
`vec_step(vector unsigned int)` `vec_step(vector signed int)` `vec_step(vector bool int)`	4
`vec_step(vector unsigned long long)` `vec_step(vector signed long long)` `vec_step(vector bool long long)`	2
`vec_step(vector float)`	4
`vec_step(vector double)`	2

Binary expressions

Some binary expressions that are used with primitive data types are extended for the vector data types.

For binary operators, each element has the operation applied to it with the same position element in the second operand. Binary operators also include assignment operators.

Table 20. Binary operators

Operator	Integer vector types	Floating-point vector types	Bool vector types
*	Yes	Yes	No
/	Yes	Yes	No
%	Yes	No	No
+	Yes	Yes	No
–	Yes	Yes	No
<<	Yes	No	No
>>	Yes	No	No
&	Yes	Yes	Yes
^	Yes	Yes	Yes
|	Yes	Yes	Yes
[]	Yes	Yes	Yes

Notes:

- The [] operator returns the vector element at the position specified.
- These operators might not be portable.

For relational operators, each element has the operation applied to it with the same position element in the second operand and the results have the AND operator applied to them to get a final result of a single value. The following table provides a summary on the binary operators that can operate on some of the vector data types.

Table 21. Relational operators

Operator	Integer vector types	Floating-point vector types	Bool vector types
==	Yes	Yes	Yes
!=	Yes	Yes	Yes
>	Yes	Yes	No
<	Yes	Yes	No
>=	Yes	Yes	No
<=	Yes	Yes	No

These operators are not supported at global scope or for objects with static duration, and there is no constant folding.

The following sections provide details on each of the supported binary operators with the vector data types. For general detailed information about binary operators, see Binary expressions in .

Related information

"Vector data types" on page 27

Assignment operator =

An assignment operator stores a value in the object designated by the left operand. If either side of an assignment expression is a vector type, both sides of the expression must be of the same vector type. Therefore, the expression a = b is valid and represents assignment if a and b are of the same vector type. Otherwise, the expression is invalid, and the compiler reports an error about inconsistent data types.

Multiplication operator *

The * (multiplication) operator yields the product of its operands.

Note: This function emulates the operation on `vector unsigned long long` and `vector signed long long`.

The following table lists the vector data types accepted as the operands, and the corresponding returned vector data types:

*Table 22. Accepted vector data types for multiplication operator **

Result types	Left operand types	Right operand types
vector unsigned char	vector unsigned char	vector unsigned char
vector signed char	vector signed char	vector signed char
vector unsigned short	vector unsigned short	vector unsigned short
vector signed short	vector signed short	vector signed short
vector unsigned int	vector unsigned int	vector unsigned int
vector signed int	vector signed int	vector signed int
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector signed long long	vector signed long long	vector signed long long
vector float	vector float	vector float
vector double	vector double	vector double

Division operator /

The / (division) operator yields the algebraic quotient of its operands.

Note: This function emulates the operation on integer vector types.

The following table lists the vector data types accepted as the operands, and the corresponding returned vector data types:

Table 23. Accepted vector data types for division operator /

Result types	Left operand types	Right operand types
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int

Table 23. Accepted vector data types for division operator / (continued)		
Result types	**Left operand types**	**Right operand types**
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

Remainder operator %

The % (remainder) operator yields the remainder from the division of the left operand by the right operand.

The following table lists the vector data types accepted as the operands, and the corresponding returned vector data types:

Table 24. Accepted vector data types for remainder operator %		
Result types	**Left operand types**	**Right operand types**
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long

Addition operator +

The + (addition) operator yields the sum of its operands.

The following table lists the vector data types accepted as the operands, and the corresponding returned vector data types:

Table 25. Accepted vector data types for addition operator +		
Result types	**Left operand types**	**Right operand types**
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

Subtraction operator -

The - (subtraction) operator yields the difference of its operands.

The following table lists the vector data types accepted as the operands, and the corresponding returned vector data types:

Table 26. Accepted vector data types for subtraction operator -

Result types	Left operand types	Right operand types
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

Bitwise left shift operator <<

The << (bitwise left shift operator) performs a left shift for each element of a vector. Each element of the result vector is the result of left shifting the corresponding element of the left operand by the number of bits specified by the value specified on the right operand, or the value of the corresponding element of the right operand, modulo the number of bits in the element. The bits that are shifted out are replaced by zeros.

The following table lists the vector data types accepted as the operands, and the corresponding returned vector data types:

Table 27. Accepted vector data types for bitwise left shift operator <<

Result types	Left operand types	Right operand types
vector unsigned char	vector unsigned char	vector unsigned char
vector signed char	vector signed char	
vector unsigned short	vector unsigned short	vector unsigned short
vector signed short	vector signed short	
vector unsigned int	vector unsigned int	vector unsigned int
vector signed int	vector signed int	
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector signed long long	vector signed long long	
vector unsigned char	vector unsigned char	unsigned long
vector signed char	vector signed char	
vector unsigned short	vector unsigned short	unsigned long
vector signed short	vector signed short	

Table 27. Accepted vector data types for bitwise left shift operator << (continued)		
Result types	**Left operand types**	**Right operand types**
vector unsigned int	vector unsigned int	unsigned long
vector signed int	vector signed int	
vector unsigned long long	vector unsigned long long	unsigned long
vector signed long long	vector signed long long	

Bitwise right shift operator >>

The >> (bitwise right shift operator) performs a logical or an algebraic right shift for each element of a vector. Each element of the result vector is the result of right shifting the corresponding element of the left operand by the number of bits specified by the value of the corresponding element of the right operand, modulo the number of bits in the element. When the right operand is an unsigned vector type, the bits that are shifted out are replaced by zeroes. While if the left operand is a signed vector type, the bits that are shifted out are replaced by copies of the most significant bit of the element of the left operand.

The following table lists the vector data types accepted as the operands, and the corresponding returned vector data types:

Table 28. Accepted vector data types for bitwise right shift operator >>		
Result types	**Left operand types**	**Right operand types**
vector unsigned char	vector unsigned char	vector unsigned char
vector unsigned short	vector unsigned short	vector unsigned short
vector unsigned int	vector unsigned int	vector unsigned int
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector unsigned char	vector unsigned char	unsigned long
vector unsigned short	vector unsigned short	unsigned long
vector unsigned int	vector unsigned int	unsigned long
vector unsigned long long	vector unsigned long long	unsigned long
vector signed char	vector signed char	vector signed char
		vector unsigned char
vector signed short	vector signed short	vector signed short
		vector unsigned short
vector signed int	vector signed int	vector signed int
		vector unsigned int
vector signed long long	vector signed long long	vector signed long long
		vector unsigned long long
vector signed char	vector signed char	unsigned long
vector signed short	vector signed short	unsigned long
vector signed int	vector signed int	unsigned long
vector signed long long	vector signed long long	unsigned long

Relational less than operator <

The < (relational less than operator) tests whether all elements of the left operand are less than the corresponding elements of the right operand. The result is 1 if all elements of the left operand are less than the corresponding elements of the right operand. Otherwise, the result is 0.

Note: A signed comparison is performed, if either of the operands is a signed integer vector.

The following table lists the vector data types accepted as the operands:

Table 29. Accepted vector data types for relational less than operator <

Left operand types	**Right operand types**
vector signed char	vector signed char
vector unsigned char	vector unsigned char
vector signed short	vector signed short
vector unsigned short	vector unsigned short
vector signed int	vector signed int
vector unsigned int	vector unsigned int
vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long
vector float	vector float
vector double	vector double

Relational greater than operator >

The > (relational greater than operator) tests whether all elements of the left operand are greater than the corresponding elements of the right operand. The result is 1 if all elements of the left operand are greater than the corresponding elements of the right operand. Otherwise, the result is 0.

Note: A signed comparison is performed, if either of the operands is a signed integer vector.

The following table lists the vector data types accepted as the operands:

Table 30. Accepted vector data types for relational greater than operator >

Left operand types	**Right operand types**
vector signed char	vector signed char
vector unsigned char	vector unsigned char
vector signed short	vector signed short
vector unsigned short	vector unsigned short
vector signed int	vector signed int
vector unsigned int	vector unsigned int
vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long
vector float	vector float
vector double	vector double

Relational less than or equal to operator <=

The <= (relational less than or equal to operator) tests whether all elements of the left operand are less than or equal to the corresponding elements of the right operand. The result is 1 if all elements of the left operand are less than or equal to the corresponding elements of the right operand. Otherwise, the result is 0.

Note: A signed comparison is performed, if either of the operands is a signed integer vector.

The following table lists the vector data types accepted as the operands:

Table 31. Accepted vector data types for relational less than or equal to operator <=

Left operand types	**Right operand types**
vector signed char	vector signed char
vector unsigned char	vector unsigned char
vector signed short	vector signed short
vector unsigned short	vector unsigned short
vector signed int	vector signed int
vector unsigned int	vector unsigned int
vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long
vector float	vector float
vector double	vector double

Relational greater than or equal to operator >=

The >= (relational greater than or equal to operator) tests whether all elements of the left operand are greater than or equal to the corresponding elements of the right operand. The result is 1 if all elements of the left operand are greater than or equal to the corresponding elements of the right operand. Otherwise, the result is 0.

Note: A signed comparison is performed, if either of the operands is a signed integer vector.

The following table lists the vector data types accepted as the operands:

Table 32. Accepted vector data types for relational greater than or equal to operator >=

Left operand types	**Right operand types**
vector signed char	vector signed char
vector unsigned char	vector unsigned char
vector signed short	vector signed short
vector unsigned short	vector unsigned short
vector signed int	vector signed int
vector unsigned int	vector unsigned int
vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long
vector float	vector float
vector double	vector double

Equality operator ==

The == (equality operator) tests whether all sets of corresponding elements of the given vectors are equal. The result is 1 if each element of the left operand is equal to the corresponding element of the right operand. Otherwise, the result is 0.

The following table lists the vector data types accepted as the operands:

Table 33. Accepted vector data types for equality operator ==

Left operand types	**Right operand types**
vector bool char	vector bool char
vector signed char	vector signed char
vector unsigned char	vector unsigned char
vector bool short	vector bool short
vector signed short	vector signed short
vector unsigned short	vector unsigned short
vector bool int	vector bool int
vector signed int	vector signed int
vector unsigned int	vector unsigned int
vector bool long long	vector bool long long
vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long
vector float	vector float
vector double	vector double

Inequality operator !=

The != (inequality operator) tests whether all sets of corresponding elements of the given vectors are not equal. The result is 1 if each element of the left operand is not equal to the corresponding element of the right operand. Otherwise, the result is 0.

The following table lists the vector data types accepted as the operands:

Table 34. Accepted vector data types for inequality operator !=

Left operand types	**Right operand types**
vector bool char	vector bool char
vector signed char	vector signed char
vector unsigned char	vector unsigned char
vector bool short	vector bool short
vector signed short	vector signed short
vector unsigned short	vector unsigned short
vector bool int	vector bool int
vector signed int	vector signed int
vector unsigned int	vector unsigned int
vector bool long long	vector bool long long

Table 34. Accepted vector data types for inequality operator != (continued)	
Left operand types	**Right operand types**
vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long
vector float	vector float
vector double	vector double

Bitwise AND operator &

The & (bitwise AND) operator compares each bit of its first operand to the corresponding bit of the second operand.

The following table lists the vector data types accepted as the operands, and the corresponding returned vector data types:

Table 35. Accepted vector data types for bitwise AND operator &		
Result types	**Left operand types**	**Right operand types**
vector bool char	vector bool char	vector bool char
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector bool short	vector bool short	vector bool short
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector bool int	vector bool int	vector bool int
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

Bitwise exclusive OR operator ^

The ^ (bitwise exclusive OR) operator compares each bit of its first operand to the corresponding bit of the second operand.

Note: `vector double` will not cause IEEE exception.

The following table lists the vector data types accepted as the operands, and the corresponding returned vector data types:

Table 36. Accepted vector data types for bitwise exclusive OR operator ^		
Result types	**Left operand types**	**Right operand types**
vector bool char	vector bool char	vector bool char
vector signed char	vector signed char	vector signed char

Table 36. Accepted vector data types for bitwise exclusive OR operator ^ (continued)		
Result types	**Left operand types**	**Right operand types**
vector unsigned char	vector unsigned char	vector unsigned char
vector bool short	vector bool short	vector bool short
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector bool int	vector bool int	vector bool int
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

Bitwise inclusive OR operator |

The | (bitwise inclusive OR) operator compares the values (in binary format) of each operand and yields a value whose bit pattern shows which bits in either of the operands has the value 1.

Note: `vector double` will not cause IEEE exception.

The following table lists the vector data types accepted as the operands, and the corresponding returned vector data types:

Table 37. Accepted vector data types for bitwise inclusive OR operator |		
Result types	**Left operand types**	**Right operand types**
vector bool char	vector bool char	vector bool char
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector bool short	vector bool short	vector bool short
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector bool int	vector bool int	vector bool int
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

Vector subscripting operator []

The [] (subscripting) operator accesses individual elements of a vector data type, similar to how array elements are accessed. The vector data type is followed by a set of square brackets containing the position of the element. The position of the first element is 0. The type of the result is the type of the elements contained in the vector type.

Note: If the position specified is outside of the valid range, the behavior is undefined.

Example:

```
vector unsigned int v1 = {1,2,3,4};
unsigned int u1, u2, u3, u4;
u1 = v1[0];     // u1=1
u2 = v1[1];     // u2=2
u3 = v1[2];     // u3=3
u4 = v1[3];     // u4=4
```

Note: You can also access and manipulate individual elements of vectors with the following intrinsic functions:

- vec_extract
- vec_insert
- vec_promote
- vec_splats

Cast expressions

The cast operator () is extended to support explicit type conversions from one vector data type to another vector data type. The exact same bit pattern is retained from the cast, and no conversion of the vector elements value takes place.

Casting between any scalar types and vector types are not allowed. To manipulate a vector element, the vector subscripting operator [], or the set of gather and scatter vector built-in functions should be used.

Compound literal expressions

A *compound literal* is a postfix expression that provides an unnamed object whose value is given by an initializer list. The C99 language feature allows you to pass parameters to functions without the need for temporary variables.

A static vector variable can be initialized with a compound literal of the same type, provided that all the initializers in the initializer list are constant expressions.

Other extensions for vector types

The following runtime library functions are also extended to support vector processing:

- `sprintf()` — Format and write data
- `sscanf()` — Read and format data
- `va_arg()`, `va_copy()`, `va_end()`, `va_start()` — Access function arguments

Vector built-in functions

Individual elements of vectors can be accessed and manipulated by using the vector built-in functions. You must enable the vector support to use these built-in functions. This section provides description of the supported vector built-in functions.

This section uses pseudo code description to represent the built-in function syntax, as shown below:

```
d = builtin_name(a, b, c)
```

In the description,

- d represents the return value of the built-in function.
- a, b, and c represent the arguments of the built-in function.
- `builtin_name` is the name of the built-in function.

For example, the syntax for the built-in function with the prototype of `vector double vec_xl(long, double*)` is represented by `d = vec_xl(a, b)`.

Allowed data types for the return value and arguments of the built-in functions are provided in the tables after the description of the built-in functions.

Note: The tables only list the supported vector data types with the `vector` keyword. The same data types with the `__vector` keyword, which is the alternative spelling of `vector`, are also supported.

Header file

To use the vector built-in functions, you must include `builtins.h` and compile the program with the LANGLVL(EXTENDED) or LANGLVL(LIBEXT) option.

Summary of vector built-in functions

The tables below summarize and categorize the vector built-in functions.

Arithmetic

Table 38. Vector built-in functions for arithmetic

Function name	Short name description	More information
vec_abs	Vector Absolute Value	See detail
vec_add_u128	Vector Add unsigned 128-bits	See detail
vec_addc	Vector Add Carryout	See detail
vec_addc_u128	Vector Add Compute Carryout unsigned 128-bits	See detail
vec_adde_u128	Vector Add With Carry unsigned 128-bits	See detail
vec_addec_u128	Vector Add With Carry Compute Carry unsigned 128-bits	See detail
vec_andc	Vector AND With Complement	See detail
vec_avg	Vector Average	See detail
vec_checksum	Vector Checksum	See detail
vec_gfmsum	Vector Galois Field Multiply Sum	See detail
vec_gfmsum_128	Vector Galois Field Multiply Sum 128-bits	See detail
vec_gfmsum_accum	Vector Galois Field Multiply Sum and Accumulate	See detail
vec_gfmsum_accum_128	Vector Galois Field Multiply Sum and Accumulate 128-bits	See detail
vec_madd	Vector Multiply Add	See detail
vec_max	Vector Maximum	See detail
vec_meadd	Vector Multiply and Add Even	See detail

Table 38. Vector built-in functions for arithmetic (continued)		
Function name	**Short name description**	**More information**
vec_mhadd	Vector Multiply and Add High	See detail
vec_min	Vector Minimum	See detail
vec_mladd	Vector Multiply and Add Low	See detail
vec_moadd	Vector Multiply and Add Odd	See detail
vec_msub	Vector Multiply Subtract	See detail
vec_msum_u128	Vector Multiply Sum Logical	See detail
vec_mule	Vector Multiply Even	See detail
vec_mulh	Vector Multiply High	See detail
vec_mulo	Vector Multiply Odd	See detail
vec_nabs	Vector Negative Absolute	See detail
vec_nmadd	Vector Negative Multiply Add	See detail
vec_nmsub	Vector Negative Multiply Subtract	See detail
vec_sqrt	Vector Square Root	See detail
vec_sub_u128	Vector Subtract unsigned 128-bits	See detail
vec_subc	Vector Subtract Carryout	See detail
vec_subc_u128	Vector Subtract Carryout unsigned 128-bits	See detail
vec_sube_u128	Vector Subtract with Carryout	See detail
vec_subec_u128	Vector Subtract with Carryout, Carryout	See detail
vec_sum_u128	Vector Sum Across Quadword	See detail
vec_sum2	Vector Sum Across Doubleword	See detail
vec_sum4	Vector Sum Across Word	See detail

Compare

Table 39. Vector built-in functions for comparing elements		
Function name	**Short name description**	**More information**
vec_cmpeq	Vector Compare Equal	See detail
vec_cmpeq_idx	Vector Compare Equal Index	See detail
vec_cmpeq_idx_cc	Vector Compare Equal Index with Condition Code	See detail
vec_cmpeq_or_0_idx	Vector Compare Equal or Zero Index	See detail
vec_cmpeq_or_0_idx_cc	Vector Compare Equal or Zero Index with Condition Code	See detail
vec_cmpge	Vector Compare Greater Than or Equal	See detail
vec_cmpgt	Vector Compare Greater Than	See detail

Table 39. Vector built-in functions for comparing elements (continued)		
Function name	**Short name description**	**More information**
vec_cmple	Vector Compare Less Than or Equal	See detail
vec_cmplt	Vector Compare Less Than	See detail
vec_cmpne_idx	Vector Compare Not Equal Index	See detail
vec_cmpne_idx_cc	Vector Compare Not Equal Index with Condition Code	See detail
vec_cmpne_or_0_idx	Vector Compare Not Equal or Zero Index	See detail
vec_cmpne_or_0_idx_cc	Vector Compare Not Equal or Zero Index with Condition Code	See detail

Compare Ranges

Table 40. Vector built-in functions for comparing ranges		
Function name	**Short name description**	**More information**
vec_cmpnrg	Vector Compare Not in Ranges	See detail
vec_cmpnrg_cc	Vector Compare Not in Ranges with Condition Code	See detail
vec_cmpnrg_idx	Vector Compare Not in Ranges Index	See detail
vec_cmpnrg_idx_cc	Vector Compare Not in Ranges Index with Condition Code	See detail
vec_cmpnrg_or_0_idx	Vector Compare Not in Ranges or Zero Index	See detail
vec_cmpnrg_or_0_idx_cc	Vector Compare Not in Ranges or Zero Index with Condition Code	See detail
vec_cmprg	Vector Compare Ranges	See detail
vec_cmprg_cc	Vector Compare Ranges with Condition Code	See detail
vec_cmprg_idx	Vector Compare Ranges Index	See detail
vec_cmprg_idx_cc	Vector Compare Ranges Index with Condition Code	See detail
vec_cmprg_or_0_idx	Vector Compare Ranges or Zero Index	See detail
vec_cmprg_or_0_idx_cc	Vector Compare Ranges or Zero Index with Condition Code	See detail

Find Any Element

Table 41. Vector built-in functions for element searching		
Function name	**Short name description**	**More information**
vec_find_any_eq	Vector Find Any Element Equal	See detail
vec_find_any_eq_cc	Vector Find Any Element Equal with Condition Code	See detail

Table 41. Vector built-in functions for element searching (continued)		
Function name	**Short name description**	**More information**
vec_find_any_eq_idx	Vector Find Any Element Equal Index	See detail
vec_find_any_eq_idx_cc	Vector Find Any Element Equal Index with Condition Code	See detail
vec_find_any_eq_or_0_idx	Vector Find Any Element Equal or Zero Index	See detail
vec_find_any_eq_or_0_idx_cc	Vector Find Any Element Equal or Zero Index with Condition Code	See detail
vec_find_any_ne	Vector Find Any Element Not Equal	See detail
vec_find_any_ne_cc	Vector Find Any Element Not Equal with Condition Code	See detail
vec_find_any_ne_idx	Vector Find Any Element Not Equal Index	See detail
vec_find_any_ne_idx_cc	Vector Find Any Element Not Equal Index with Condition Code	See detail
vec_find_any_ne_or_0_idx	Vector Find Any Element Not Equal or Zero Index	See detail
vec_find_any_ne_or_0_idx_cc	Vector Find Any Element Not Equal or Zero Index with Condition Code	See detail

Gather and Scatter

Table 42. Vector built-in functions for gathering and scattering elements		
Function name	**Short name description**	**More information**
vec_bperm_u128	Vector Bit Permute	See detail
vec_extract	Vector Extract	See detail
vec_gather_element	Vector Gather Element	See detail
vec_insert	Vector Insert	See detail
vec_insert_and_zero	Vector Insert and Zero	See detail
vec_perm	Vector Permute	See detail
vec_promote	Vector Promote	See detail
vec_scatter_element	Vector Scatter Element	See detail
vec_sel	Vector Select	See detail

Generate Mask

Table 43. Vector built-in functions for generating mask		
Function name	**Short name description**	**More information**
vec_genmask	Vector Generate Byte Mask	See detail
vec_genmasks_8	Vector Generate Mask (Byte)	See detail

Table 43. Vector built-in functions for generating mask (continued)		
Function name	**Short name description**	**More information**
vec_genmasks_16	Vector Generate Mask (Halfword)	See detail
vec_genmasks_32	Vector Generate Mask (Word)	See detail
vec_genmasks_64	Vector Generate Mask (Doubleword)	See detail

Copy until Zero

Table 44. Vector built-in functions for copying until a zero is encountered		
Function name	**Short name description**	**More information**
vec_cp_until_zero	Vector Copy Until Zero	See detail
vec_cp_until_zero_cc	Vector Copy Until Zero	See detail

Load and Store

Table 45. Vector built-in functions for loading and storing vectors		
Function name	**Short name description**	**More information**
vec_load_bndry	Vector Load to Block Boundary	See detail
vec_load_len	Vector Load with Length	See detail
vec_load_len_r	Vector Load Rightmost with Length	See detail
vec_load_pair	Vector Load Pair	See detail
vec_store_len	Vector Store with Length	See detail
vec_store_len_r	Vector Store Rightmost with Length	See detail
vec_xl	Vector Load	See detail
vec_xst	Vector Store	See detail

Logical

Table 46. Vector built-in functions for logical calculation		
Function name	**Short name description**	**More information**
vec_cntlz	Vector Count Leading Zeros	See detail
vec_cnttz	Vector Count Trailing Zeros	See detail
vec_eqv	Vector XNOR	See detail
vec_nand	Vector NAND	See detail
vec_nor	Vector NOR	See detail
vec_orc	Vector OR with Complement	See detail
vec_popcnt	Vector Population Count	See detail

Merge

Table 47. Vector built-in functions for merging vectors		
Function name	**Short name description**	**More information**
vec_mergeh	Vector Merge High	See detail
vec_mergel	Vector Merge Low	See detail

Pack and Unpack

Table 48. Vector built-in functions for pack and unpack		
Function name	**Short name description**	**More information**
vec_pack	Vector Pack	See detail
vec_packs	Vector Pack Saturate	See detail
vec_packs_cc	Vector Pack Saturate Condition Code	See detail
vec_packsu	Vector Pack Saturated Unsigned	See detail
vec_packsu_cc	Vector Pack Saturated Unsigned Condition Code	See detail
vec_unpackh	Vector Unpack High Element	See detail
vec_unpackl	Vector Unpack Low Element	See detail

Replicate

Table 49. Vector built-in functions for replicating vector elements		
Function name	**Short name description**	**More information**
vec_splat	Vector Splat	See detail
vec_splat_s8	Vector Splat Signed Byte	See detail
vec_splat_s16	Vector Splat Signed Halfword	See detail
vec_splat_s32	Vector Splat Signed Word	See detail
vec_splat_s64	Vector Splat Signed Doubleword	See detail
vec_splat_u8	Vector Splat Unsigned Byte	See detail
vec_splat_u16	Vector Splat Unsigned Halfword	See detail
vec_splat_u32	Vector Splat Unsigned Word	See detail
vec_splat_u64	Vector Splat Doubleword	See detail
vec_splats	Vector Splats	See detail

Rotate and Shift

Table 50. Vector built-in functions for rotate and shift

Function name	Short name description	More information
vec_rl	Vector Element Rotate Left	See detail
vec_rl_mask	Vector Element Rotate and Insert Under Mask	See detail
vec_rli	Vector Element Rotate Left Immediate	See detail
vec_slb	Vector Shift Left by Byte	See detail
vec_sld	Vector Shift Left Double by Byte	See detail
vec_sldw	Vector Shift Left Double by Word	See detail
vec_sll	Vector Shift Left	See detail
vec_srab	Vector Shift Right Arithmetic by Byte	See detail
vec_sral	Vector Shift Right Arithmetic	See detail
vec_srb	Vector Shift Right by Byte	See detail
vec_srl	Vector Shift Right	See detail

Rounding and Conversion

Table 51. Vector built-in functions for rounding and conversion

Function name	Short name description	More information
vec_ceil	Vector Ceiling	See detail
vec_double	Vector Convert from Logical	See detail
vec_doublee	Vector Load Lengthened	See detail
vec_extend_s64	Vector Sign Extend to Doubleword	See detail
vec_floate	Vector Load Rounded	See detail
vec_floor	Vector Floor	See detail
vec_rint	Vector Round to Integer	See detail
vec_round	Vector Round to Nearest	See detail
vec_roundc	Vector Round to Current®	See detail
vec_roundm	Vector Round toward Negative Infinity	See detail
vec_roundp	Vector Round toward Positive Infinity	See detail
vec_roundz	Vector Round toward Zero	See detail
vec_signed	Vector Convert double to signed long long	See detail
vec_trunc	Vector Truncate	See detail
vec_unsigned	Vector Convert double to unsigned long long	See detail

Test

Table 52. Vector built-in functions for testing

Function name	Short name description	More information
vec_fp_test_data_class	Vector Floating-Point Test Data Class	See detail
vec_test_mask	Vector Test under Mask	See detail

All Predicates

Table 53. Vector built-in functions for searching and comparing all elements

Function name	Short name description	More information
vec_all_eq	All Elements Equal	See detail
vec_all_ge	All Elements Greater Than or Equal	See detail
vec_all_gt	All Elements Greater Than	See detail
vec_all_le	All Elements Less Than or Equal	See detail
vec_all_lt	All Elements Less Than	See detail
vec_all_nan	All Elements Not a Number	See detail
vec_all_ne	All Elements Not Equal	See detail
vec_all_nge	All Elements Not Greater Than or Equal	See detail
vec_all_ngt	All Elements Not Greater Than	See detail
vec_all_nle	All Elements Not Less Than or Equal	See detail
vec_all_nlt	All Elements Not Less Than	See detail
vec_all_numeric	All Elements Numeric	See detail

Any Predicates

Table 54. Vector built-in functions for searching and comparing any elements

Function name	Short name description	More information
vec_any_eq	Any Element Equal	See detail
vec_any_ge	Any Element Greater Than or Equal	See detail
vec_any_gt	Any Element Greater Than	See detail
vec_any_le	Any Element Less Than or Equal	See detail
vec_any_lt	Any Element Less Than	See detail
vec_any_nan	Any Element Not a Number	See detail
vec_any_ne	Any Element Not Equal	See detail
vec_any_nge	Any Element Not Greater Than or Equal	See detail
vec_any_ngt	Any Element Not Greater Than	See detail
vec_any_nle	Any Element Not Less Than or Equal	See detail

Table 54. Vector built-in functions for searching and comparing any elements (continued)		
Function name	**Short name description**	**More information**
vec_any_nlt	Any Element Not Less Than	See detail
vec_any_numeric	Any Element Numeric	See detail

Arithmetic

This section describes built-in functions for arithmetic.

vec_abs: Vector Absolute Value

```
d = vec_abs(a)
```

Returns a vector containing the absolute values of the contents of the given vector. The value of each element of the result is the absolute value of the corresponding element of a.

Note: `vector float` and `vector double` will not cause IEEE exception.

Table 55. Vector Absolute Value		
d	**a**	**MIN ARCH**
vector signed char	vector signed char	ARCH(11) 1
vector signed short	vector signed short	ARCH(11) 1
vector signed int	vector signed int	ARCH(11) 1
vector signed long long	vector signed long long	ARCH(11) 1
vector float	vector float	ARCH(12) 1
vector double	vector double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_add_u128: Vector Add unsigned 128-bits

```
d = vec_add_u128(a, b)
```

Adds unsigned quadword values.

The function operates on vectors as 128-bit unsigned integers. It returns low 128 bits of a + b.

Table 56. Vector Add unsigned 128-bits			
d	**a**	**b**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)

vec_addc: Vector Add Carryout

```
d = vec_addc(a, b)
```

Returns a vector containing the carry produced by adding each set of corresponding elements of the given vectors.

Each resulting element is set to 1 if there is a carry, and 0 otherwise.

Table 57. Vector Add Carryout			
d	**a**	**b**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11) 1
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(11)
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_addc_u128: Vector Add Compute Carryout unsigned 128-bits

```
d = vec_addc_u128(a, b)
```

Gets the carry bit of the 128-bit addition of two quadword values.

This function operates on the vectors as 128-bit unsigned integers. It returns the carry out of a + b.

If there is a carry on the addition, the bit 127 of d is set to 1; otherwise it is set to 0. All other bits of d are 0.

Table 58. Vector Add Compute Carryout unsigned 128-bits			
d	**a**	**b**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)

vec_adde_u128: Vector Add With Carry unsigned 128-bits

```
d = vec_adde_u128(a, b, c)
```

Adds unsigned quadword values with carry bit from the previous operation.

This function operates on the vectors as 128-bit unsigned integers. It returns low 128 bits of a + b + (c & 1).

Note: Only the carry bit (127-bit) of c is used, and the other bits are ignored.

Table 59. Vector Add With Carry unsigned 128-bits				
d	**a**	**b**	**c**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)

vec_addec_u128: Vector Add With Carry Compute Carry unsigned 128-bits

```
d = vec_addec_u128(a, b, c)
```

Gets the carry bit of the 128-bit addition of two quadword values with carry bit from a previous operation.

This function operates on the vectors as 128-bit unsigned integers. It returns the carry out of a + b + (c & 1).

If there is a carry on this addition, the 127-bit of d is 1, otherwise 0. All other bits of d are 0.

Note: Only the carry bit (127-bit) of c is used, and the other bits are ignored.

Table 60. Vector Add With Carry Compute Carry unsigned 128-bits				
d	**a**	**b**	**c**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)

vec_avg: Vector Average

```
d = vec_avg(a, b)
```

Returns a vector containing the average of each set of the corresponding elements of the given vectors.

Table 61. Vector Average			
d	**a**	**b**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	ARCH(11) 1
vector signed short	vector signed short	vector signed short	ARCH(11) 1
vector signed int	vector signed int	vector signed int	ARCH(11) 1
vector signed long long	vector signed long long	vector signed long long	ARCH(11)
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11) 1
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11) 1
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11) 1
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(11)
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_checksum: Vector Checksum

```
d = vec_checksum(a, b)
```

Returns a vector with the 1-indexed element containing a checksum computed from the summation of all vector elements in a and the 1-indexed element of b. All other vector elements will be 0.

Table 62. Vector Checksum			
d	**a**	**b**	**MIN ARCH**
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11)

vec_gfmsum: Vector Galois Field Multiply Sum

```
d = vec_gfmsum(a, b)
```

Performs a Galois field multiply sum on each element of the given vectors.

Each element of a is multiplied in a Galois field with the corresponding element of b. The Galois field has an order of two. This multiplication is similar to standard binary multiplication, but instead of adding the shifted multiplicand it is exclusive ORed. The resulting even-odd pairs of double element-sized products are exclusive ORed with each other and placed in the corresponding double-wide element of the returned vector.

Table 63. Vector Galois Field Multiply Sum			
d	**a**	**b**	**MIN ARCH**
vector unsigned short	vector unsigned char	vector unsigned char	ARCH(11)
vector unsigned int	vector unsigned short	vector unsigned short	ARCH(11)
vector unsigned long long	vector unsigned int	vector unsigned int	ARCH(11)

vec_gfmsum_128: Vector Galois Field Multiply Sum 128-bits

```
d = vec_gfmsum_128(a, b)
```

Performs a Galois field multiply sum on the 2 elements of the given vectors.

Each element of a is multiplied in a Galois field with the corresponding element of b. The Galois field has an order of two. This multiplication is similar to standard binary multiplication, but instead of adding the shifted multiplicand it is exclusive ORed. The resulting 128-bits products are exclusive ORed with each other and return as a vector unsigned char.

Table 64. Vector Galois Field Multiply Sum 128-bits			
d	**a**	**b**	**MIN ARCH**
vector unsigned char	vector unsigned long long	vector unsigned long long	ARCH(11)

vec_gfmsum_accum: Vector Galois Field Multiply Sum and Accumulate

```
d = vec_gfmsum_accum(a, b, c)
```

Performs a Galois field multiply sum and accumulate on each element of the given vectors.

Each element of a is multiplied in a Galois field with the corresponding element of b. The Galois field has an order of two. This multiplication is similar to standard binary multiplication, but instead of adding the shifted multiplicand it is exclusive ORed. The resulting even-odd pairs of double element-sized products are exclusive ORed with each other and exclusive ORed with the corresponding double-wide element of c, and returned by the function.

Table 65. Vector Galois Field Multiply Sum and Accumulate				
d	**a**	**b**	**c**	**MIN ARCH**
vector unsigned short	vector unsigned char	vector unsigned char	vector unsigned short	ARCH(11)
vector unsigned int	vector unsigned short	vector unsigned short	vector unsigned int	ARCH(11)
vector unsigned long long	vector unsigned int	vector unsigned int	vector unsigned long long	ARCH(11)

vec_gfmsum_accum_128: Vector Galois Field Multiply Sum and Accumulate 128-bits

```
d = vec_gfmsum_accum_128(a, b, c)
```

Performs a Galois field multiply sum and accumulate on the 2 elements of the given vectors.

Each element of a is multiplied in a Galois field with the corresponding element of b. The Galois field has an order of two. This multiplication is similar to standard binary multiplication, but instead of adding the shifted multiplicand it is exclusive ORed. The resulting 128-bits products are exclusive ORed with each other and exclusive ORed with the 128-bits c, and returned by the function.

Table 66. Vector Galois Field Multiply Sum and Accumulate 128-bits				
d	**a**	**b**	**c**	**MIN ARCH**
vector unsigned char	vector unsigned long long	vector unsigned long long	vector unsigned char	ARCH(11)

vec_madd: Vector Multiply Add

```
d = vec_madd(a, b, c)
```

Returns a vector containing the results of performing a fused multiply-add operation for each corresponding set of elements of the given vectors. The value of each element of the result is the product of the values of the corresponding elements of a and b, added to the value of the corresponding element of c.

Table 67. Vector Multiply Add				
d	**a**	**b**	**c**	**MIN ARCH**
vector float	vector float	vector float	vector float	ARCH(12) 1
vector double	vector double	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.				

vec_max: Vector Maximum

```
d = vec_max(a, b)
```

Returns a vector containing the maximum value from each set of corresponding elements of the given vectors. The value of each element of the result is the maximum of the values of the corresponding elements of a and b.

This function emulates the operation on `vector double` under ARCH(11).

Note: The emulation is done as `(a>b)?a:b`, which is sightly different from the IEEE semantics. For details, see the results tables for performing the "C-Style Max Macro" and IEEE MaxNum on the VECTOR FP MAXIMUM (VFMAX) instruction in z/Architecture Principles of Operation.

Table 68. Vector Maximum			
d	**a**	**b**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	ARCH(11) 1
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11) 1
vector signed short	vector signed short	vector signed short	ARCH(11) 1
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11) 1
vector signed int	vector signed int	vector signed int	ARCH(11) 1
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11) 1
vector signed long long	vector signed long long	vector signed long long	ARCH(11) 1
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(11) 1
vector float	vector float	vector float	ARCH(12) 1

Table 68. Vector Maximum (continued)			
d	**a**	**b**	**MIN ARCH**
vector double	vector double	vector double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_meadd: Vector Multiply and Add Even

```
d = vec_meadd(a, b, c)
```

Returns a vector containing double element-sized results of performing a multiply-and-add operation for each of the even-indexed elements on the given vectors. The value of each element of d is the result of adding the corresponding element of c to the double element-sized product of the even-indexed elements of a and b.

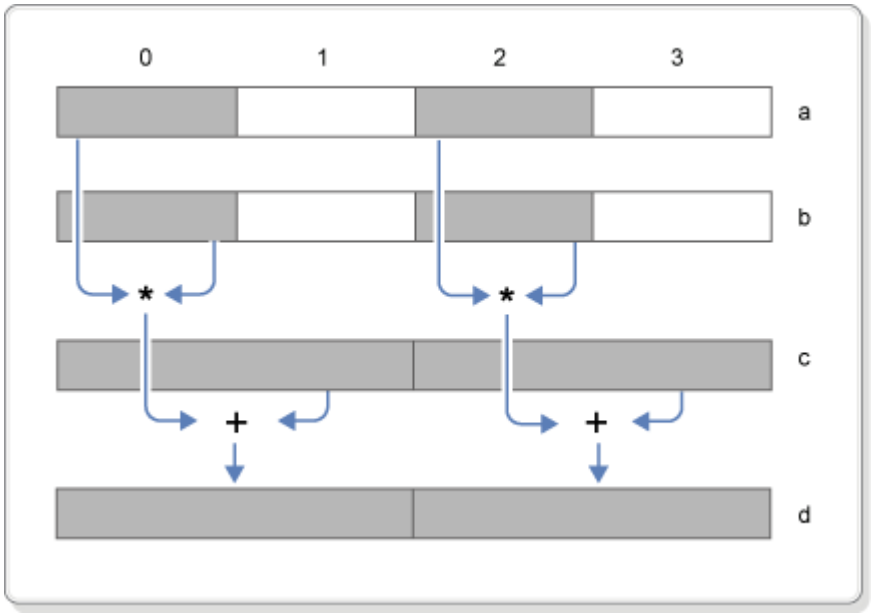


Figure 7. Multiply and add even of integer elements (32-bit)

Table 69. Vector Multiply and Add Even				
d	**a**	**b**	**c**	**MIN ARCH**
vector unsigned short	vector unsigned char	vector unsigned char	vector unsigned short	ARCH(11)
vector signed short	vector signed char	vector signed char	vector signed short	ARCH(11)
vector unsigned int	vector unsigned short	vector unsigned short	vector unsigned int	ARCH(11)
vector signed int	vector signed short	vector signed short	vector signed int	ARCH(11)
vector unsigned long long	vector unsigned int	vector unsigned int	vector unsigned long long	ARCH(11)
vector signed long long	vector signed int	vector signed int	vector signed long long	ARCH(11)

vec_mhadd: Vector Multiply and Add High

```
d = vec_mhadd(a, b, c)
```

Returns a vector containing the most significant ("high") half of the double element-sized results of performing a multiply-and-add operation for each corresponding set of elements of the given vectors. The value of each element of the result is the value of the most significant half of the double element-sized of the product of the values of the corresponding elements of a and b, added to the value of the corresponding element of c.

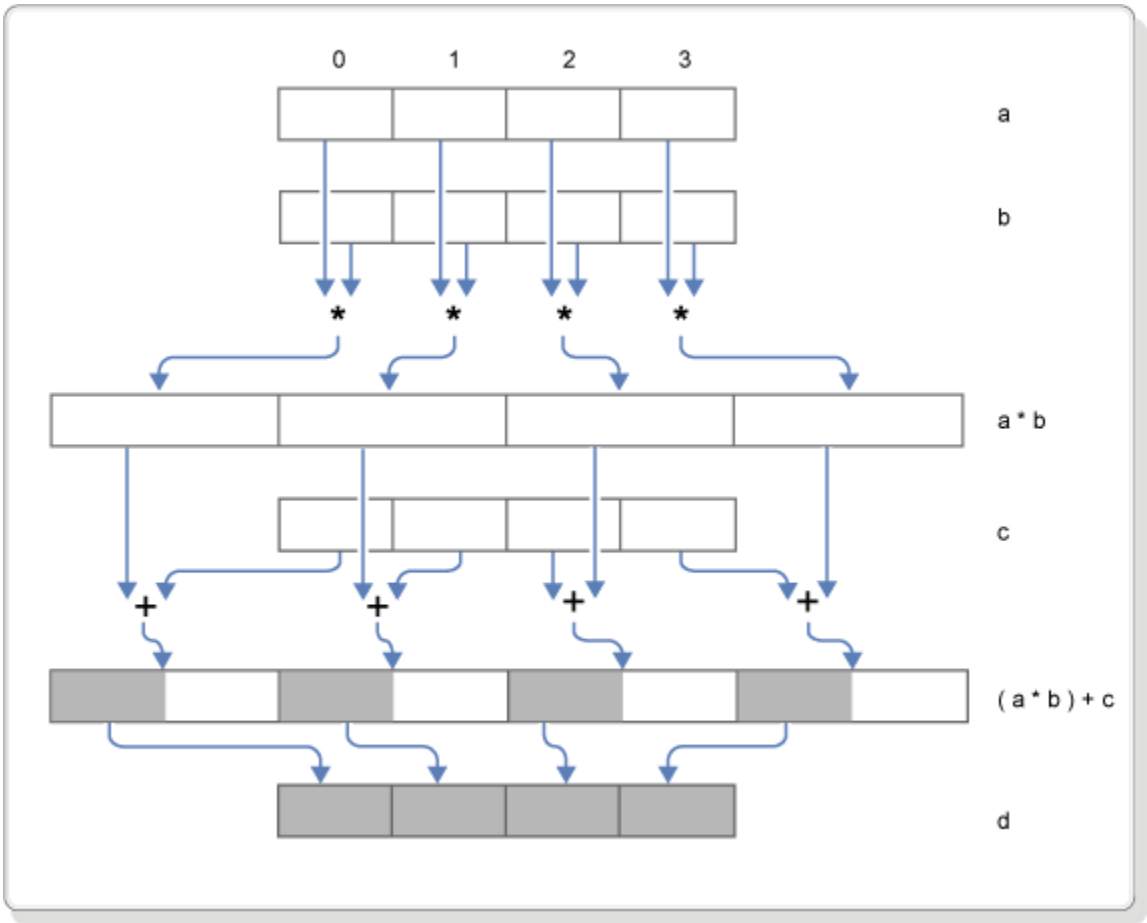


Figure 8. Multiply and add high of integer elements (32-bit)

Table 70. Vector Multiply and Add High

d	a	b	c	MIN ARCH
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)
vector signed char	vector signed char	vector signed char	vector signed char	ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11)
vector signed short	vector signed short	vector signed short	vector signed short	ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11)
vector signed int	vector signed int	vector signed int	vector signed int	ARCH(11)

vec_min: Vector Minimum

```
d = vec_min(a, b)
```

Returns a vector containing the minimum value from each set of corresponding elements of the given vectors. The value of each element of the result is the minimum of the values of the corresponding elements of a and b.

This function emulates the operation on `vector double` under ARCH(11).

Note: The emulation is done as `!(b<a)?a:b`, which is sightly different from the IEEE semantics. For details, see the results tables for performing the IEEE MaxNum on the VECTOR FP MAXIMUM (VFMAX) instruction in z/Architecture Principles of Operation.

Table 71. Vector Minimum

d	a	b	MIN ARCH
vector signed char	vector signed char	vector signed char	ARCH(11) 1
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11) 1
vector signed short	vector signed short	vector signed short	ARCH(11) 1
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11) 1
vector signed int	vector signed int	vector signed int	ARCH(11) 1
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11) 1
vector signed long long	vector signed long long	vector signed long long	ARCH(11) 1
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(11) 1
vector float	vector float	vector float	ARCH(12) 1
vector double	vector double	vector double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_mladd: Vector Multiply and Add Low

```
d = vec_mladd(a, b, c)
```

Returns a vector containing the least significant ("low") half of the double element-sized results of performing a multiply-and-add operation for each corresponding set of elements of the given vectors. The value of each element of the result is the value of the least significant half of the double element-sized of the product of the values of the corresponding elements of a and b, added to the value of the corresponding element of c.

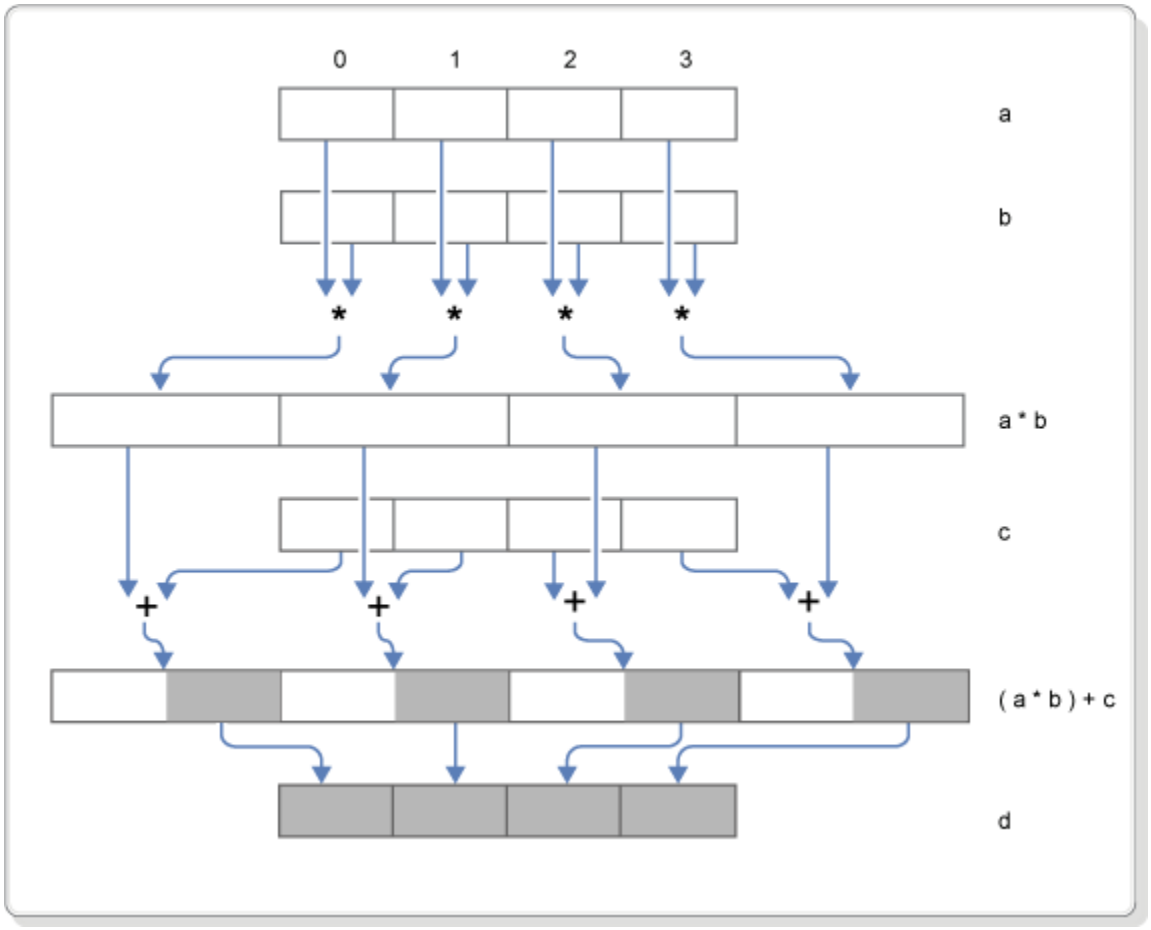


Figure 9. Multiply and add low of integer elements (32-bit)

Table 72. Vector Multiply and Add Low				
d	**a**	**b**	**c**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)
vector signed char	vector unsigned char	vector signed char	vector signed char	ARCH(11)
	vector signed char	vector unsigned char	vector unsigned char	ARCH(11)
	vector signed char	vector signed char	vector signed char	ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11) 1
vector signed short	vector unsigned short	vector signed short	vector signed short	ARCH(11) 1
	vector signed short	vector unsigned short	vector unsigned short	ARCH(11) 1
	vector signed short	vector signed short	vector signed short	ARCH(11) 1
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11)
vector signed int	vector unsigned int	vector signed int	vector signed int	ARCH(11)
	vector signed int	vector unsigned int	vector unsigned int	ARCH(11)
	vector signed int	vector signed int	vector signed int	ARCH(11)

Table 72. Vector Multiply and Add Low (continued)				
d	**a**	**b**	**c**	**MIN ARCH**
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.				

vec_moadd: Vector Multiply and Add Odd

```
d = vec_moadd(a, b, c)
```

Returns a vector containing a double element-sized results of performing a multiply-and-add operation for each of the odd-indexed elements on the given vectors. The value of each element is the value of the double element-sized of the product of the values of the odd-indexed elements of a and b, added to the value of the corresponding element of c.

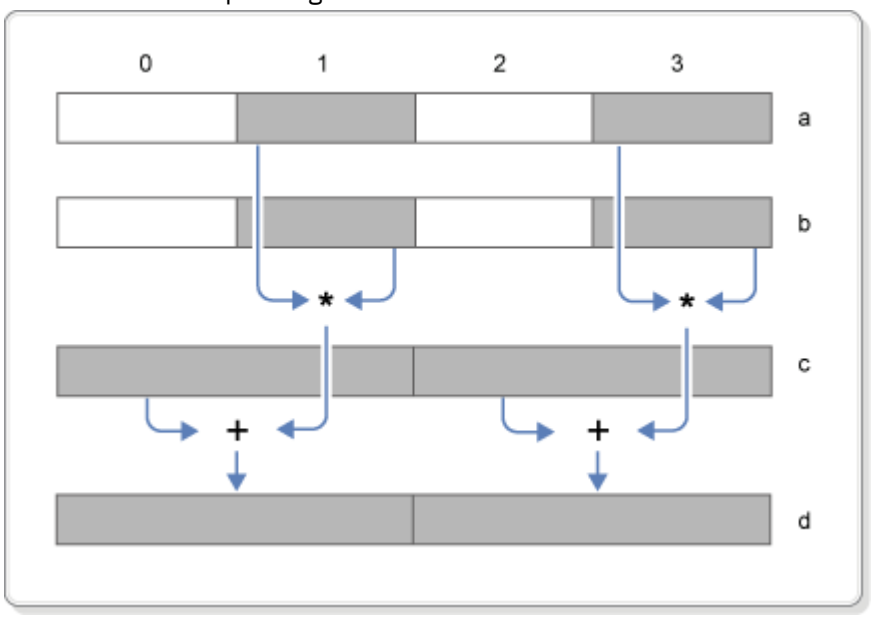


Figure 10. Multiply and add odd of integer elements (32-bit)

Table 73. Vector Multiply and Add Odd				
d	**a**	**b**	**c**	**MIN ARCH**
vector unsigned short	vector unsigned char	vector unsigned char	vector unsigned short	ARCH(11)
vector signed short	vector signed char	vector signed char	vector signed short	ARCH(11)
vector unsigned int	vector unsigned short	vector unsigned short	vector unsigned int	ARCH(11)
vector signed int	vector signed short	vector signed short	vector signed int	ARCH(11)
vector unsigned long long	vector unsigned int	vector unsigned int	vector unsigned long long	ARCH(11)
vector signed long long	vector signed int	vector signed int	vector signed long long	ARCH(11)

vec_msub: Vector Multiply Subtract

```
d = vec_msub(a, b, c)
```

Returns a vector containing the results of performing a multiply-subtract operation using the given vectors. This function multiplies each element in a by the corresponding element in b, and then subtracts the corresponding element in c from the result.

Table 74. Vector Multiply Subtract

d	**a**	**b**	**c**	**MIN ARCH**
vector float	vector float	vector float	vector float	ARCH(12) 1
vector double	vector double	vector double	vector double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_msum_u128: Vector Multiply Sum Logical

```
e = vec_msum_u128(a, b, c, d)
```

Returns a vector that contains a 128-bit unsigned integer, which is the sum of the following values:

- The 128-bit product of the 0-index elements of vector a and b. If d equals 8 or 12, the product is multiplied by 2. If d equals 0, the product remains the same.
- The 128-bit product of the 1-index elements of vector a and b. If d equals 4 or 12, the product is multiplied by 2. If d equals 0, the product remains the same.
- Vector c as a 128-bit unsigned integer.

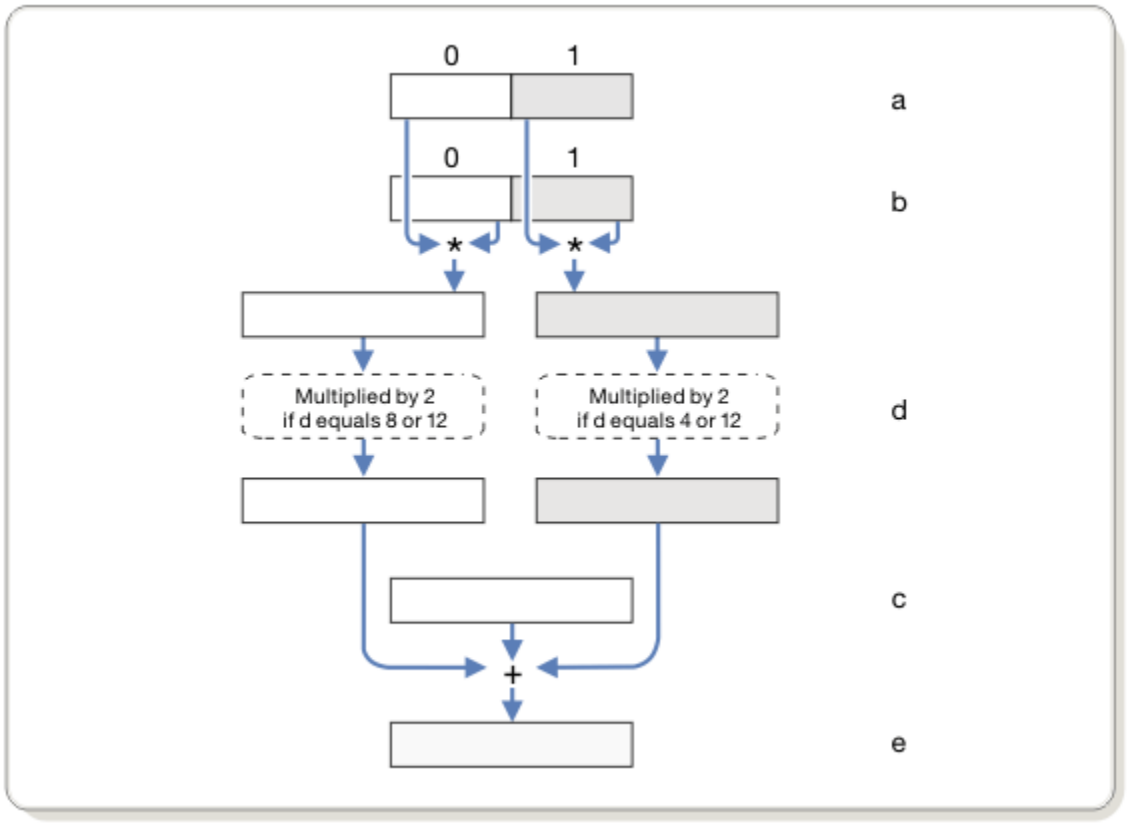


Figure 11. Multiply and sum with optional multiplication by 2

Table 75. Vector Multiply Sum Logical					
e	**a**	**b**	**c**	**d**	**MIN ARCH**
vector unsigned char	vector unsigned long long	vector unsigned long long	vector unsigned char	0, 4, 8, 12	ARCH(12)

vec_mule: Vector Multiply Even

```
d = vec_mule(a, b)
```

Returns a vector containing the results of performing a multiply operation for each corresponding set of even-indexed elements of the given vectors, and extended to double element size.

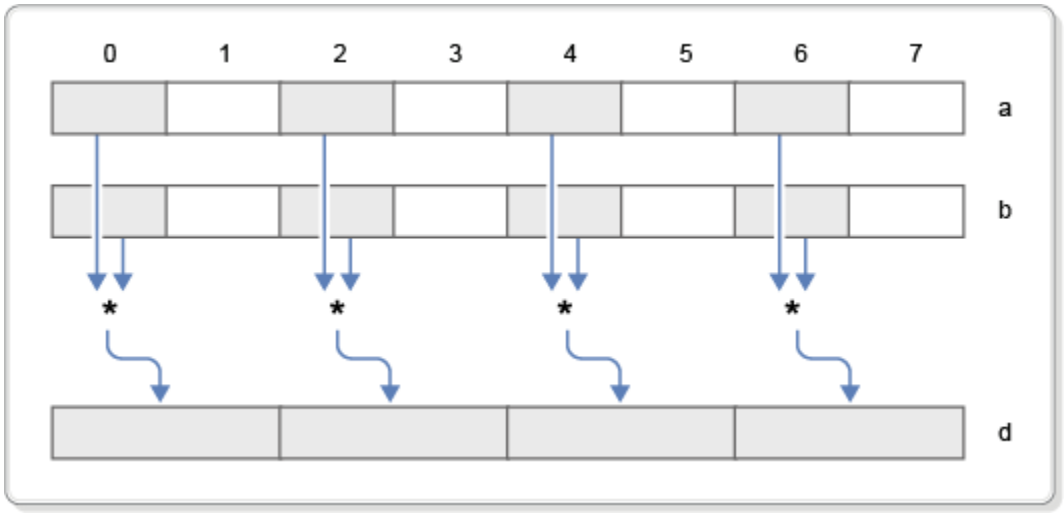


Figure 12. Even multiply of 4 integer elements (16-bit)

Table 76. Vector Multiply Even

d	a	b	MIN ARCH
vector unsigned short	vector unsigned char	vector unsigned char	ARCH(11) 1
vector signed short	vector signed char	vector signed char	ARCH(11) 1
vector unsigned int	vector unsigned short	vector unsigned short	ARCH(11) 1
vector signed int	vector signed short	vector signed short	ARCH(11) 1
vector unsigned long long	vector unsigned int	vector unsigned int	ARCH(11) 1
vector signed long long	vector signed int	vector signed int	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_mulh: Vector Multiply High

```
d = vec_mulh(a, b)
```

Returns a vector containing the most significant ("high") half of results of performing a multiply operation using the given vectors. This function multiplies corresponding elements in the given vectors, the most significant half of the double element-sized product is assigned to the result of the corresponding elements in the result vector.

Table 77. Vector Multiply High

d	a	b	MIN ARCH
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)
vector signed char	vector signed char	vector signed char	ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11)
vector signed short	vector signed short	vector signed short	ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11)
vector signed int	vector signed int	vector signed int	ARCH(11)

vec_mulo: Vector Multiply Odd

```
d = vec_mulo(a, b)
```

Returns a vector containing the results of performing a multiply operation for each corresponding set of odd-indexed elements of the given vectors, and extended to double element size.

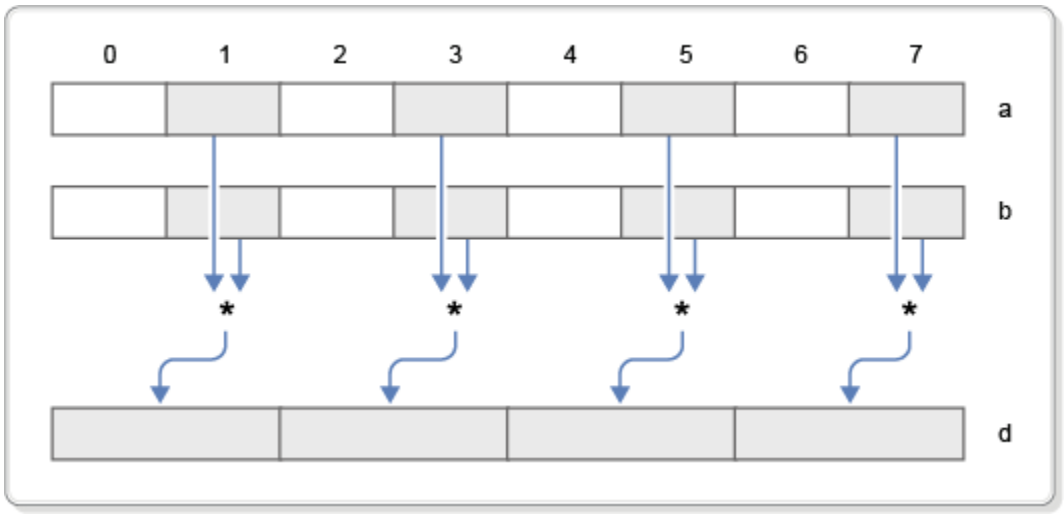


Figure 13. Odd multiply of 4 integer elements (16-bit)

Table 78. Vector Multiply Odd

d	a	b	MIN ARCH
vector unsigned short	vector unsigned char	vector unsigned char	ARCH(11) 1
vector signed short	vector signed char	vector signed char	ARCH(11) 1
vector unsigned int	vector unsigned short	vector unsigned short	ARCH(11) 1
vector signed int	vector signed short	vector signed short	ARCH(11) 1
vector unsigned long long	vector unsigned int	vector unsigned int	ARCH(11) 1
vector signed long long	vector signed int	vector signed int	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_nabs: Vector Negative Absolute

```
d = vec_nabs(a)
```

Returns a vector containing the results of performing a negative-absolute operation using the given vector. This function computes the absolute value of each element in the given vector and then assigns the negated value of the result to the corresponding elements in the result vector.

Note: This built-in function will not cause IEEE exception.

Table 79. Vector Negative Absolute

d	a	MIN ARCH
vector float	vector float	ARCH(12) 1
vector double	vector double	ARCH(11) 1

Table 79. Vector Negative Absolute (continued)		
d	**a**	**MIN ARCH**
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.		

vec_nmadd: Vector Negative Multiply Add

```
d = vec_nmadd(a, b, c)
```

Returns a vector containing the results of performing a negative multiply-add operation on the given vectors. The value of each element of d is the product of the corresponding elements of a and b, added to the corresponding elements of c, and then multiplied by -1.0.

Table 80. Vector Negative Multiply Add				
d	**a**	**b**	**c**	**MIN ARCH**
vector float	vector float	vector float	vector float	ARCH(12) 1
vector double	vector double	vector double	vector double	ARCH(12) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.				

vec_nmsub: Vector Negative Multiply Subtract

```
d = vec_nmsub(a, b, c)
```

Returns a vector containing the results of performing a negative multiply-subtract operation on the given vectors. The value of each element of d is the product of the corresponding elements of a and b, subtracted by the corresponding elements of c, and then multiplied by -1.0.

Table 81. Vector Negative Multiply Subtract				
d	**a**	**b**	**c**	**MIN ARCH**
vector float	vector float	vector float	vector float	ARCH(12) 1
vector double	vector double	vector double	vector double	ARCH(12) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.				

vec_sqrt: Vector Square Root

```
d = vec_sqrt(a)
```

Returns a vector containing the square root of each element in the given vector.

Table 82. Vector Square Root		
d	**a**	**MIN ARCH**
vector float	vector float	ARCH(12) 1

Table 82. Vector Square Root (continued)		
d	**a**	**MIN ARCH**
vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.		

vec_sub_u128: Vector Subtract unsigned 128-bits

```
d = vec_sub_u128(a, b)
```

Subtracts unsigned quadword values.

This function operates on the vectors as 128-bit unsigned integers. It returns low 128 bits of a - b.

Table 83. Vector Subtract unsigned 128-bits			
d	**a**	**b**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)

vec_subc: Vector Subtract Carryout

```
d = vec_subc(a, b)
```

Returns a vector containing the borrow produced by subtracting each of corresponding elements of b from a.

On each resulting element, the value is 0 if a borrow occurred, or 1 if no borrow occurred.

Table 84. Vector Subtract Carryout			
d	**a**	**b**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11) 1
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(11)
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_subc_u128: Vector Subtract Carryout unsigned 128-bits

```
d = vec_subc_u128(a, b)
```

Gets the carry bit of the 128-bit subtraction of two quadword values.

This function operates on the vectors as 128-bit unsigned integers. It returns a vector containing the borrow produced by subtracting b from a, as unsigned 128-bits integers.

If no borrow occurred, the bit 127 of d is 1; otherwise it is set to 0. All other bits of d are 0.

Table 85. Vector Subtract Carryout unsigned 128-bits			
d	**a**	**b**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)

vec_sube_u128: Vector Subtract with Carryout

```
d = vec_sube_u128(a, b, c)
```

Subtracts unsigned quadword values with carry bit from a previous operation.

This function operates on the vectors as 128-bit unsigned integers. It returns a vector containing the result of subtracting of b from a, and the carryout bit from a previous operation.

Note: Only the borrow indication bit (127-bit) of c is used, and the other bits are ignored.

Table 86. Vector Subtract with Carryout				
d	**a**	**b**	**c**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)

vec_subec_u128: Vector Subtract with Carryout, Carryout

```
d = vec_subec_u128(a, b, c)
```

Gets the carry bit of the 128-bit subtraction of two quadword values with carry bit from the previous operation.

It returns a vector containing the carryout produced from the result of subtracting of b from a, and the carryout bit from a previous operation. If no borrow occurred, the 127-bit of d is 1, otherwise 0. All other bits of d are 0.

Note: Only the borrow indication bit (127-bit) of c is used, and the other bits are ignored.

Table 87. Vector Subtract with Carryout, Carryout				
d	**a**	**b**	**c**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)

vec_sum_u128: Vector Sum Across Quadword

```
d = vec_sum_u128(a, b)
```

Returns a vector containing the results of performing a sum across all the elements in each of the quadword of vector a, and the rightmost word or doubleword element of the b. The result is an unsigned 128-bit integer. The result vector is obtained as follow:

For `vector unsigned int` operands:

```
d = a[0] + a[1] + a[2] + a[3] + b[3]
```

For `vector unsigned long long` operands:

```
d = a[0] + a[1] + b[1]
```

Table 88. Vector Sum Across Quadword			
d	**a**	**b**	**MIN ARCH**
vector unsigned char	vector unsigned int	vector unsigned int	ARCH(11)
	vector unsigned long long	vector unsigned long long	ARCH(11)

vec_sum2: Vector Sum Across Doubleword

```
d = vec_sum2(a, b)
```

Returns a vector containing the results of performing a sum across all the elements in each of the doubleword of vector a, and the rightmost sub-element of the corresponding doubleword of b.

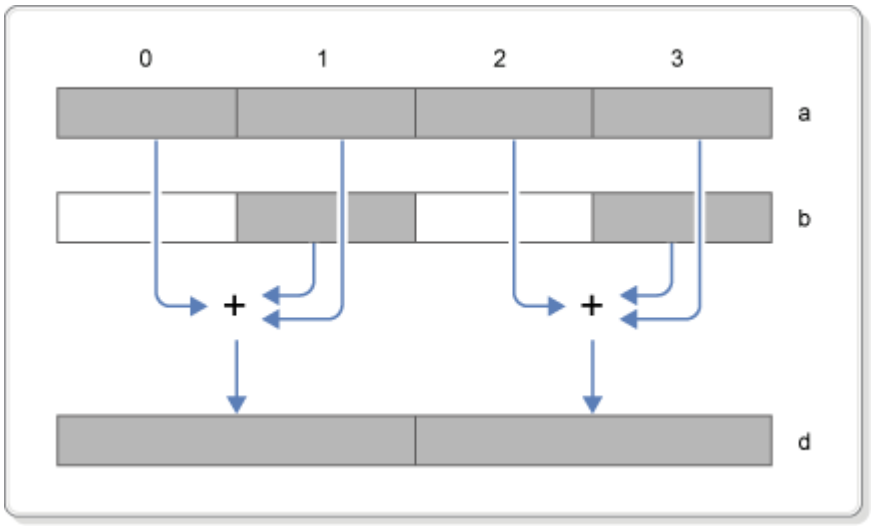


Figure 14. Sum across doubleword of unsigned integer elements (32-bit)

The result vector is obtained as follows:

For `vector unsigned short` operands:

```
d[0] = a[0] + a[1] + a[2] + a[3] + b[3]
d[1] = a[4] + a[5] + a[6] + a[7] + b[7]
```

For `vector unsigned int` operands:

```
d[0] = a[0] + a[1] + b[1]
d[1] = a[2] + a[3] + b[3]
```

Table 89. Vector Sum Across Doubleword

d	a	b	MIN ARCH
vector unsigned long long	vector unsigned short	vector unsigned short	ARCH(11)
	vector unsigned int	vector unsigned int	ARCH(11)

vec_sum4: Vector Sum Across Word

```
d = vec_sum4(a, b)
```

Returns a vector containing the results of performing a sum across all the elements in each of the word of vector a, and the rightmost sub-element of the corresponding word of b.

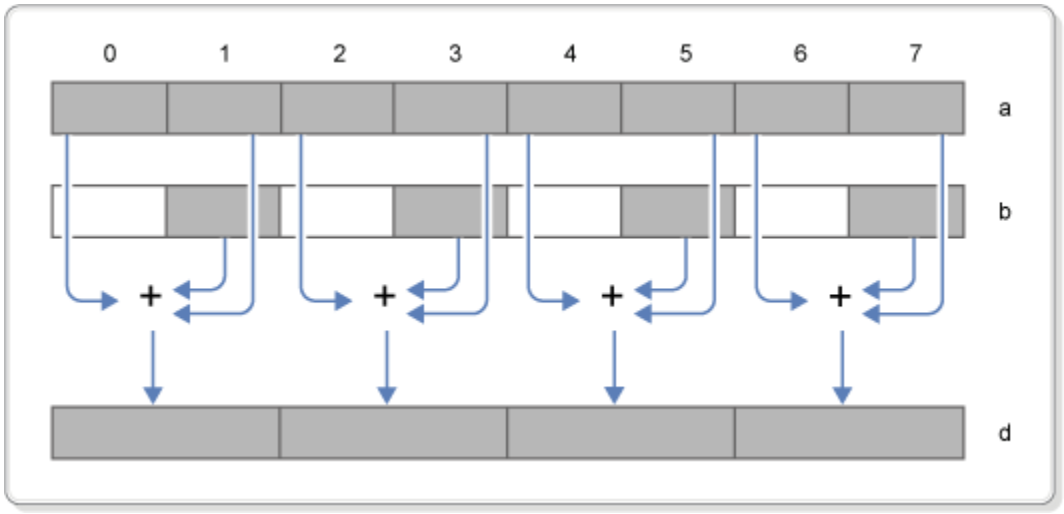


Figure 15. Sum across word of unsigned short elements (16-bit)

The result vector is obtained as follow:

For `vector unsigned char` operands:

```
d[0] = a[0] + a[1] + a[2] + a[3] + b[3]
d[1] = a[4] + a[5] + a[6] + a[7] + b[7]
d[2] = a[8] + a[8] + a[10] + a[11] + b[11]
d[3] = a[12] + a[13] + a[14] + a[15] + b[15]
```

For `vector unsigned short` operands:

```
d[0] = a[0] + a[1] + b[1]
d[1] = a[2] + a[3] + b[3]
d[2] = a[4] + a[5] + b[5]
d[3] = a[6] + a[7] + b[7]
```

Table 90. Vector Sum Across Word

d	a	b	MIN ARCH
vector unsigned int	vector unsigned char	vector unsigned char	ARCH(11)
	vector unsigned short	vector unsigned short	ARCH(11)

Compare

This section describes vector built-in functions for comparing elements.

vec_cmpeq: Vector Compare Equal

```
d = vec_cmpeq(a, b)
```

Returns a vector containing the results of comparing each set of corresponding elements of the given vectors for equality. For each element of the result, the value of each bit is 1 if the corresponding elements of a and b are equal. Otherwise, the value of each bit is 0.

Table 91. Vector Compare Equal

d	a	b	MIN ARCH
vector bool char	vector bool char	vector bool char	ARCH(11) 1
	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1

Table 91. Vector Compare Equal (continued)			
d	**a**	**b**	**MIN ARCH**
vector bool short	vector bool short	vector bool short	ARCH(11) 1
	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
vector bool int	vector bool int	vector bool int	ARCH(11) 1
	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
vector bool long long	vector bool long long	vector bool long long	ARCH(11) 1
	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

Note: `vec_cmpne` is available on the XL C/C++ compilers for some other platforms. You can define the following macro to migrate programs from other platforms to the Enterprise Metal C for z/OS compiler.

```
#define vec_cmpne(a, b) (~vec_cmpeq(a,b))
```

vec_cmpeq_idx: Vector Compare Equal Index

```
d = vec_cmpeq_idx(a, b)
```

Returns the lowest byte-index of comparing each set of corresponding elements of the given vectors for equality. If the two vectors are not equal, the result is 16.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 92. Vector Compare Equal Index			
d	**a**	**b**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	ARCH(11)
vector unsigned char	vector bool char	vector bool char	ARCH(11)
	vector unsigned char	vector unsigned char	ARCH(11)
vector signed short	vector signed short	vector signed short	ARCH(11)
vector unsigned short	vector bool short	vector bool short	ARCH(11)
	vector unsigned short	vector unsigned short	ARCH(11)
vector signed int	vector signed int	vector signed int	ARCH(11)
vector unsigned int	vector bool int	vector bool int	ARCH(11)
	vector unsigned int	vector unsigned int	ARCH(11)

vec_cmpeq_idx_cc: Vector Compare Equal Index with Condition Code

```
d = vec_cmpeq_idx_cc(a, b, c)
```

Returns the lowest byte-index of comparing each set of corresponding elements of the given vectors for equality. If the two vectors are not equal, the result is 16. c is set to 1, if there is any elements of a equals the corresponding element of b, otherwise c is set to 3.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 93. Vector Compare Equal Index with Condition Code

d	**a**	**b**	**c**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	int *	ARCH(11)
vector unsigned char	vector bool char	vector bool char		ARCH(11)
	vector unsigned char	vector unsigned char		ARCH(11)
vector signed short	vector signed short	vector signed short		ARCH(11)
vector unsigned short	vector bool short	vector bool short		ARCH(11)
	vector unsigned short	vector unsigned short		ARCH(11)
vector signed int	vector signed int	vector signed int		ARCH(11)
vector unsigned int	vector bool int	vector bool int		ARCH(11)
	vector unsigned int	vector unsigned int		ARCH(11)

vec_cmpeq_or_0_idx: Vector Compare Equal or Zero Index

```
d = vec_cmpeq_or_0_idx(a, b)
```

Returns the lowest byte-index of comparing each set of corresponding elements of the given vectors for equality, and comparing each elements of a against 0. If the two vectors are not equal, and no elements of a is 0, the result is 16.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 94. Vector Compare Equal or Zero Index

d	**a**	**b**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	ARCH(11)
vector unsigned char	vector bool char	vector bool char	ARCH(11)
	vector unsigned char	vector unsigned char	ARCH(11)
vector signed short	vector signed short	vector signed short	ARCH(11)
vector unsigned short	vector bool short	vector bool short	ARCH(11)
	vector unsigned short	vector unsigned short	ARCH(11)
vector signed int	vector signed int	vector signed int	ARCH(11)
vector unsigned int	vector bool int	vector bool int	ARCH(11)
	vector unsigned int	vector unsigned int	ARCH(11)

vec_cmpeq_or_0_idx_cc: Vector Compare Equal or Zero Index with Condition Code

```
d = vec_cmpeq_or_0_idx_cc(a, b, c)
```

Returns the lowest byte-index of comparing each set of corresponding elements of the given vectors for equality, and comparing each elements of a against 0. If the two vectors are not equal, and no elements of a is 0, the result is 16.

c would be set to one of the following values:

- 0 - if no elements of the 2 vectors are equal, and at least one element from a with a value of 0.
- 1 - if at least one element of a equals the corresponding element of b, and no elements of a has a value of 0.
- 2 - if at least one element of a equals the corresponding element of b with an equal value, and there is at least one element from a has a value of 0.
- 3 - if no element of a equals the corresponding element of b, and there is no element from a with a value of 0.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 95. Vector Compare Equal or Zero Index with Condition Code

d	**a**	**b**	**c**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	int *	ARCH(11)
vector unsigned char	vector bool char	vector bool char		ARCH(11)
	vector unsigned char	vector unsigned char		ARCH(11)
vector signed short	vector signed short	vector signed short		ARCH(11)
vector unsigned short	vector bool short	vector bool short		ARCH(11)
	vector unsigned short	vector unsigned short		ARCH(11)
vector signed int	vector signed int	vector signed int		ARCH(11)
vector unsigned int	vector bool int	vector bool int		ARCH(11)
	vector unsigned int	vector unsigned int		ARCH(11)

vec_cmpge: Vector Compare Greater Than or Equal

```
d = vec_cmpge(a, b)
```

Returns a vector containing the results of a greater-than-or-equal-to comparison between each set of corresponding elements of the given vectors. For each element of the result, the value of each bit is 1 if the value of the corresponding element of a is greater than or equal to the value of the corresponding element of b. Otherwise, the value of each bit is 0.

This function emulates the operation on the integer vectors.

Table 96. Vector Compare Greater Than or Equal

d	**a**	**b**	**MIN ARCH**
vector bool char	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
vector bool short	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
vector bool int	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector float	vector float	ARCH(12) 1

Table 96. Vector Compare Greater Than or Equal (continued)			
d	**a**	**b**	**MIN ARCH**
vector bool long long	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector double	vector double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_cmpgt: Vector Compare Greater Than

```
d = vec_cmpgt(a, b)
```

Returns a vector containing the results of a greater-than comparison between each set of corresponding elements of the given vectors. For each element of the result, the value of each bit is 1 if the value of the corresponding element of a is greater than the value of the corresponding element of b. Otherwise, the value of each bit is 0.

Table 97. Vector Compare Greater Than			
d	**a**	**b**	**MIN ARCH**
vector bool char	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
vector bool short	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
vector bool int	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
vector bool long long	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector double	vector double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_cmple: Vector Compare Less Than or Equal

```
d = vec_cmple(a, b)
```

Returns a vector containing the results of a less-than-or-equal-to comparison between each set of corresponding elements of the given vectors. For each element of the result, the value of each bit is 1 if the value of the corresponding element of a is less than or equal to the value of the corresponding element of b. Otherwise, the value of each bit is 0.

Table 98. Vector Compare Less Than or Equal			
d	**a**	**b**	**MIN ARCH**
vector bool char	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
vector bool short	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
vector bool int	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
vector bool long long	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_cmplt: Vector Compare Less Than

```
d = vec_cmplt(a, b)
```

Returns a vector containing the results of a less-than comparison between each set of corresponding elements of the given vectors. For each element of the result, the value of each bit is 1 if the value of the corresponding element of a is less than the value of the corresponding element of b. Otherwise, the value of each bit is 0.

This operation emulates the operation on the integer vectors.

Table 99. Vector Compare Less Than			
d	**a**	**b**	**MIN ARCH**
vector bool char	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
vector bool short	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
vector bool int	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
vector bool long long	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector double	vector double	ARCH(11) 1

Table 99. Vector Compare Less Than (continued)			
d	**a**	**b**	**MIN ARCH**
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_cmpne_idx: Vector Compare Not Equal Index

```
d = vec_cmpne_idx(a, b)
```

Returns the lowest byte-index of comparing each set of corresponding elements of the given vectors for inequality. If the two vectors are equal, the result is 16.

The result is placed into byte element seven of the returned vector, all other bytes are set to 0.

Table 100. Vector Compare Not Equal Index			
d	**a**	**b**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	ARCH(11)
vector unsigned char	vector bool char	vector bool char	ARCH(11)
	vector unsigned char	vector unsigned char	ARCH(11)
vector signed short	vector signed short	vector signed short	ARCH(11)
vector unsigned short	vector bool short	vector bool short	ARCH(11)
	vector unsigned short	vector unsigned short	ARCH(11)
vector signed int	vector signed int	vector signed int	ARCH(11)
vector unsigned int	vector bool int	vector bool int	ARCH(11)
	vector unsigned int	vector unsigned int	ARCH(11)

vec_cmpne_idx_cc: Vector Compare Not Equal Index with Condition Code

```
d = vec_cmpne_idx_cc(a, b, c)
```

Returns the lowest byte-index of comparing each set of corresponding elements of the given vectors for inequality. If the two vectors are equal, the result is 16.

c is set to the following value:

- 1 - if there is a mismatch and that first element from the 0-index of a is less than the corresponding element of b.
- 2 - if there is a mismatch and that element from the 0-index of a is greater than the corresponding element of b.
- 3 - if the two vectors are equal.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 101. Vector Compare Not Equal Index with Condition Code				
d	**a**	**b**	**c**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	int *	ARCH(11)
vector unsigned char	vector bool char	vector bool char		ARCH(11)
	vector unsigned char	vector unsigned char		ARCH(11)
vector signed short	vector signed short	vector signed short		ARCH(11)
vector unsigned short	vector bool short	vector bool short		ARCH(11)
	vector unsigned short	vector unsigned short		ARCH(11)
vector signed int	vector signed int	vector signed int		ARCH(11)
vector unsigned int	vector bool int	vector bool int		ARCH(11)
	vector unsigned int	vector unsigned int		ARCH(11)

vec_cmpne_or_0_idx: Vector Compare Not Equal or Zero Index

```
d = vec_cmpne_or_0_idx(a, b)
```

Returns the lowest byte-index of comparing each set of corresponding elements of the given vectors for inequality, and comparing each elements of a against 0. If the two vectors are equal, and no elements of a is 0, the result is 16.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 102. Vector Compare Not Equal or Zero Index			
d	**a**	**b**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	ARCH(11)
vector unsigned char	vector bool char	vector bool char	ARCH(11)
	vector unsigned char	vector unsigned char	ARCH(11)
vector signed short	vector signed short	vector signed short	ARCH(11)
vector unsigned short	vector bool short	vector bool short	ARCH(11)
	vector unsigned short	vector unsigned short	ARCH(11)
vector signed int	vector signed int	vector signed int	ARCH(11)
vector unsigned int	vector bool int	vector bool int	ARCH(11)
	vector unsigned int	vector unsigned int	ARCH(11)

vec_cmpne_or_0_idx_cc: Vector Compare Not Equal or Zero Index with Condition Code

```
d = vec_cmpne_or_0_idx_cc(a, b, c)
```

Returns the lowest byte-index of comparing each set of corresponding elements of the given vectors for inequality, and comparing each elements of a against 0. If the two vectors are equal, and no elements of a is 0, the result is 16.

c is set to the following value:

- 0 - if zero is found on an element of a, starting from the 0-index, before there is a mismatch between the corresponding elements of a and b.

- 1 - if there is a mismatch, and that first element, from the 0-index, of a is less than the corresponding element of b, and prior to the mismatch a is not 0.
- 2 - if there is a mismatch, and that element, from the 0-index, of a is greater than the corresponding element of b, and prior to the mismatch a is not 0.
- 3 - if the two vectors are equal, and there is no element from a with a value of 0.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 103. Vector Compare Not Equal or Zero Index with Condition Code

d	a	b	c	MIN ARCH
vector signed char	vector signed char	vector signed char	int *	ARCH(11)
vector unsigned char	vector bool char	vector bool char		ARCH(11)
	vector unsigned char	vector unsigned char		ARCH(11)
vector signed short	vector signed short	vector signed short		ARCH(11)
vector unsigned short	vector bool short	vector bool short		ARCH(11)
	vector unsigned short	vector unsigned short		ARCH(11)
vector signed int	vector signed int	vector signed int		ARCH(11)
vector unsigned int	vector bool int	vector bool int		ARCH(11)
	vector unsigned int	vector unsigned int		ARCH(11)

Compare Ranges

This section describes vector built-in functions for comparing ranges.

vec_cmpnrg: Vector Compare Not in Ranges

```
d = vec_cmpnrg(a, b, c)
```

Check if each element of a is not within any of the ranges specified by b and c. Each even-odd element pairs of b define values for the limits of the ranges. The corresponding even-odd pairs of elements in c control how the comparison to be performed, in the following way:

Comparison	for vector unsigned char	for vector unsigned short	for vector unsigned int
ignore - result of comparison always TRUE	0	0	0
equal	0x80	0x8000	0x80000000
not equal	0x60	0x6000	0x60000000
greater than	0x20	0x2000	0x20000000
greater than or equal	0xA0	0xA000	0xA0000000
less than	0x40	0x4000	0x40000000
less than and equal	0xC0	0xC000	0xC0000000
force to FALSE	0xE0	0xE000	0xE0000000

For each element of the result, the value of each bit is 1 if the corresponding element of a was not contained in any of the specified ranges. Otherwise, the value of each bit is 0.

Table 104. Vector Compare Not in Ranges				
d	**a**	**b**	**c**	**MIN ARCH**
vector bool char	vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)
vector bool short	vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11)
vector bool int	vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11)

Example 1: Comparing 2 ranges

```
vector unsigned int a = {11, 22, 33, 44};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x20000000,
                         0x40000000}; // {GT, LT, GT, LT}

vector bool int d = vec_cmpnrg(a, b, c);
// d = {0, 0xFFFFFFFF, 0, 0xFFFFFFFF}
```

In this example, each element of a is checked to be (> 10 AND < 20) OR (> 30 AND < 40).

Example 2: Comparing a single range, and a specific value

```
vector unsigned int a = {11, 22, 33, 30};
vector unsigned int b = {10, 20, 30, 30};
vector unsigned int c = {0x20000000, 0x40000000, 0x80000000,
                         0x80000000}; // {GT, LT, EQ, EQ}

vector bool int d = vec_cmpnrg(a, b, c);
// d = {0, 0xFFFFFFFF, 0xFFFFFFFF, 0)
```

In this example, each element of a is checked to be (> 10 AND < 20) OR equals to 30.

Example 3: Comparing a single range

```
vector unsigned int a = {11, 22, 33, 44};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x00000000,
                         0x00000000}; // {GT, LT, X, X}

vector bool int d = vec_cmpnrg(a, b, c);
// d = {0, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF}
```

In this example, each element of a is checked to be (> 10 AND < 20) only.

vec_cmpnrg_cc: Vector Compare Not in Ranges with Condition Code

```
e = vec_cmpnrg_cc(a, b, c, d)
```

Check if each element of a is not within any of the ranges specified by b and c. Each even-odd element pairs of b define values for the limits of the ranges. The corresponding even-odd pairs of elements in c control how the comparison to be performed, in the following way:

Comparison	**for vector unsigned char**	**for vector unsigned short**	**for vector unsigned int**
ignore - result of comparison always TRUE	0	0	0
equal	0x80	0x8000	0x80000000
not equal	0x60	0x6000	0x60000000
greater than	0x20	0x2000	0x20000000
greater than or equal	0xA0	0xA000	0xA0000000

Comparison	for vector unsigned char	for vector unsigned short	for vector unsigned int
less than	0x40	0x4000	0x40000000
less than and equal	0xC0	0xC000	0xC0000000
force to FALSE	0xE0	0xE000	0xE0000000

For each element of the result, the value of each bit is 1 if the corresponding element of a was not contained in any of the specified ranges. Otherwise, the value of each bit is 0.

d is set to 1, if there is at least one element of a is found not within any of the ranges. Otherwise, d is set to 3.

Table 105. Vector Compare Not in Ranges with Condition Code

e	a	b	c	d	MIN ARCH
vector bool char	vector unsigned char	vector unsigned char	vector unsigned char	int *	ARCH(11)
vector bool short	vector unsigned short	vector unsigned short	vector unsigned short		ARCH(11)
vector bool int	vector unsigned int	vector unsigned int	vector unsigned int		ARCH(11)

vec_cmpnrg_idx: Vector Compare Not in Ranges Index

```
d = vec_cmpnrg_idx(a, b, c)
```

Returns the lowest byte-index of the element of a that is not within any of the ranges specified by b and c. Each even-odd element pairs of b define values for the limits of the ranges. The corresponding even-odd pairs of elements in c control how the comparison to be performed, in the following way:

Comparison	for vector unsigned char	for vector unsigned short	for vector unsigned int
ignore - result of comparison always TRUE	0	0	0
equal	0x80	0x8000	0x80000000
not equal	0x60	0x6000	0x60000000
greater than	0x20	0x2000	0x20000000
greater than or equal	0xA0	0xA000	0xA0000000
less than	0x40	0x4000	0x40000000
less than and equal	0xC0	0xC000	0xC0000000
force to FALSE	0xE0	0xE000	0xE0000000

The result is the lowest byte-index from element of a that is not contained in any of the specified ranges. Otherwise, the result is 16.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 106. Vector Compare Not in Ranges Index

d	a	b	c	MIN ARCH
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)

Table 106. Vector Compare Not in Ranges Index (continued)				
d	**a**	**b**	**c**	**MIN ARCH**
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11)

Example:

```
vector unsigned int a = {1, 11, 22, 33};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x20000000,
                         0x40000000}; // {GT, LT, GT, LT}

vector unsigned int d = vec_cmpnrg_idx(a, b, c); // byte 7 of d = 0
```

In this example, each element of a is tested to be NOT((> 10 AND < 20) OR (>30 AND < 40)), the first element (byte index 0) is the first element satisfying the condition.

vec_cmpnrg_idx_cc: Vector Compare Not in Ranges Index with Condition Code

```
e = vec_cmpnrg_idx_cc(a, b, c, d)
```

Returns the lowest byte-index of the element of a that is not within any of the ranges specified by b and c. Each even-odd element pairs of b define values for the limits of the ranges. The corresponding even-odd pairs of elements in c control how the comparison to be performed, in the following way:

Comparison	**for vector unsigned char**	**for vector unsigned short**	**for vector unsigned int**
ignore - result of comparison always TRUE	0	0	0
equal	0x80	0x8000	0x80000000
not equal	0x60	0x6000	0x60000000
greater than	0x20	0x2000	0x20000000
greater than or equal	0xA0	0xA000	0xA0000000
less than	0x40	0x4000	0x40000000
less than and equal	0xC0	0xC000	0xC0000000
force to FALSE	0xE0	0xE000	0xE0000000

d is set to 1, if there is at least one element of a is found not within any of the ranges. Otherwise, d is set to 3.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 107. Vector Compare Not in Ranges Index with Condition Code					
e	**a**	**b**	**c**	**d**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	int *	ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short		ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int		ARCH(11)

vec_cmpnrg_or_0_idx: Vector Compare Not in Ranges or Zero Index

```
d = vec_cmpnrg_or_0_idx(a, b, c)
```

Returns the lowest byte-index of the element of a that is 0 or not within any of the ranges specified by b and c. Each even-odd element pairs of b define values for the limits of the ranges. The corresponding even-odd pairs of elements in c control how the comparison to be performed, in the following way:

Comparison	for vector unsigned char	for vector unsigned short	for vector unsigned int
ignore - result of comparison always TRUE	0	0	0
equal	0x80	0x8000	0x80000000
not equal	0x60	0x6000	0x60000000
greater than	0x20	0x2000	0x20000000
greater than or equal	0xA0	0xA000	0xA0000000
less than	0x40	0x4000	0x40000000
less than and equal	0xC0	0xC000	0xC0000000
force to FALSE	0xE0	0xE000	0xE0000000

The result is the lowest byte-index from element of a that is 0 or not contained in any of the specified ranges. Otherwise, the result is 16.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 108. Vector Compare Not in Ranges or Zero Index

d	a	b	c	MIN ARCH
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11)

Example:

```
vector unsigned int a = {11, 33, 0, 22};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x20000000,
                         0x40000000}; // {GT, LT, GT, LT}

vector unsigned int d = vec_cmpnrg_or_0_idx(a, b, c); // byte 7 of d = 8
```

In this example, each element of a is tested to be (equals 0) OR NOT((> 10 AND < 20) OR (>30 AND < 40)), the third element (byte index 8) is the first element satisfying the condition.

vec_cmpnrg_or_0_idx_cc: Vector Compare Not in Ranges or Zero Index with Condition Code

```
e = vec_cmpnrg_or_0_idx_cc(a, b, c, d)
```

Returns the lowest byte-index of the element of a that is 0 or not within any of the ranges specified by b and c. Each even-odd element pairs of b define values for the limits of the ranges. The corresponding even-odd pairs of elements in c control how the comparison to be performed, in the following way:

Comparison	for vector unsigned char	for vector unsigned short	for vector unsigned int
ignore - result of comparison always TRUE	0	0	0
equal	0x80	0x8000	0x80000000
not equal	0x60	0x6000	0x60000000
greater than	0x20	0x2000	0x20000000
greater than or equal	0xA0	0xA000	0xA0000000
less than	0x40	0x4000	0x40000000
less than and equal	0xC0	0xC000	0xC0000000
force to FALSE	0xE0	0xE000	0xE0000000

d is set to one of the following:

- 0 - if 0 was found on an element of a, before an element was not found within the specified range.
- 1 - if no element of a is 0, and there is at least one element of a found not in any of the ranges.
- 2 - if 0 was found on an element of a after an element was found not within the specified range.
- 3 - no element is 0 and found to be not within any of the specified range.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 109. Vector Compare Not in Ranges or Zero Index with Condition Code

e	a	b	c	d	MIN ARCH
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	int *	ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short		ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int		ARCH(11)

vec_cmprg: Vector Compare Ranges

```
d = vec_cmprg(a, b, c)
```

Check if each element of a is within any of the ranges specified by b and c. Each even-odd element pairs of b define values for the limits of the ranges. The corresponding even-odd pairs of elements in c control how the comparison to be performed, in the following way:

Comparison	for vector unsigned char	for vector unsigned short	for vector unsigned int
ignore - result of comparison always TRUE	0	0	0
equal	0x80	0x8000	0x80000000
not equal	0x60	0x6000	0x60000000
greater than	0x20	0x2000	0x20000000
greater than or equal	0xA0	0xA000	0xA0000000

Comparison	for vector unsigned char	for vector unsigned short	for vector unsigned int
less than	0x40	0x4000	0x40000000
less than and equal	0xC0	0xC000	0xC0000000
force to FALSE	0xE0	0xE000	0xE0000000

For each element of the result, the value of each bit is 1 if the corresponding element of a was contained in any of the specified ranges. Otherwise, the value of each bit is 0.

Table 110. Vector Compare Ranges

d	a	b	c	MIN ARCH
vector bool char	vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)
vector bool short	vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11)
vector bool int	vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11)

Example 1: Comparing 2 ranges

```
vector unsigned int a = {11, 22, 33, 44};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x20000000,
                         0x40000000}; // {GT, LT, GT, LT}

vector bool int d = vec_cmprg(a, b, c);
// d = {0xFFFFFFFF, 0, 0xFFFFFFFF, 0}
```

In this example, each element of a is checked to be (> 10 AND < 20) OR (> 30 AND < 40).

Example 2: Comparing a single range, and a specific value

```
vector unsigned int a = {11, 22, 33, 30};
vector unsigned int b = {10, 20, 30, 30};
vector unsigned int c = {0x20000000, 0x40000000, 0x80000000,
                         0x80000000}; // {GT, LT, EQ, EQ}

vector bool int d = vec_cmprg(a, b, c);
// d = {0xFFFFFFFF, 0, 0, 0xFFFFFFFF)
```

In this example, each element of a is checked to be (> 10 AND < 20) OR equals to 30.

Example 3: Comparing a single range

```
vector unsigned int a = {11, 22, 33, 44};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x00000000,
                         0x00000000}; // {GT, LT, X, X}

vector bool int d = vec_cmprg(a, b, c);
// d = {0xFFFFFFFF, 0, 0, 0}
```

In this example, each element of a is checked to be (> 10 AND < 20) only.

vec_cmprg_cc: Vector Compare Ranges with Condition Code

```
e = vec_cmprg_cc(a, b, c, d)
```

Check if each element of a is within any of the ranges specified by b and c. Each even-odd element pairs of b define values for the limits of the ranges. The corresponding even-odd pairs of elements in c control how the comparison to be performed, in the following way:

Comparison	for vector unsigned char	for vector unsigned short	for vector unsigned int
ignore - result of comparison always TRUE	0	0	0
equal	0x80	0x8000	0x80000000
not equal	0x60	0x6000	0x60000000
greater than	0x20	0x2000	0x20000000
greater than or equal	0xA0	0xA000	0xA0000000
less than	0x40	0x4000	0x40000000
less than and equal	0xC0	0xC000	0xC0000000
force to FALSE	0xE0	0xE000	0xE0000000

For each element of the result, the value of each bit is 1 if the corresponding element of a was contained in any of the specified ranges. Otherwise, the value of each bit is 0.

d is set to 1, if there is at least one element of a found in any of the ranges. Otherwise, d is set to 3.

Table 111. Vector Compare Ranges with Condition Code

e	a	b	c	d	MIN ARCH
vector bool char	vector unsigned char	vector unsigned char	vector unsigned char	int *	ARCH(11)
vector bool short	vector unsigned short	vector unsigned short	vector unsigned short		ARCH(11)
vector bool int	vector unsigned int	vector unsigned int	vector unsigned int		ARCH(11)

vec_cmprg_idx: Vector Compare Ranges Index

```
d = vec_cmprg_idx(a, b, c)
```

Returns the lowest byte-index of the element of a that is within any of the ranges specified by b and c. Each even-odd element pairs of b define values for the limits of the ranges. The corresponding even-odd pairs of elements in c control how the comparison to be performed, in the following way:

Comparison	for vector unsigned char	for vector unsigned short	for vector unsigned int
ignore - result of comparison always TRUE	0	0	0
equal	0x80	0x8000	0x80000000
not equal	0x60	0x6000	0x60000000
greater than	0x20	0x2000	0x20000000
greater than or equal	0xA0	0xA000	0xA0000000
less than	0x40	0x4000	0x40000000
less than and equal	0xC0	0xC000	0xC0000000
force to FALSE	0xE0	0xE000	0xE0000000

The result is the lowest byte-index from element of a that is contained in any of the specified ranges. Otherwise, the result is 16.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 112. Vector Compare Ranges Index

d	a	b	c	MIN ARCH
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11)

Example:

```
vector unsigned int a = {1, 11, 22, 33};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x20000000,
                         0x40000000}; // {GT, LT, GT, LT}

vector unsigned int d = vec_cmprg_idx(a, b, c); // byte 7 of d = 4
```

In this example, each element of a is tested to be ((> 10 AND < 20) OR (>30 AND < 40)), the second element (byte index 4) is the first element satisfying the condition.

vec_cmprg_idx_cc: Vector Compare Ranges Index with Condition Code

```
e = vec_cmprg_idx_cc(a, b, c, d)
```

Returns the lowest byte-index of the element of a that is within any of the ranges specified by b and c. Each even-odd element pairs of b define values for the limits of the ranges. The corresponding even-odd pairs of elements in c control how the comparison to be performed, in the following way:

Comparison	for vector unsigned char	for vector unsigned short	for vector unsigned int
ignore - result of comparison always TRUE	0	0	0
equal	0x80	0x8000	0x80000000
not equal	0x60	0x6000	0x60000000
greater than	0x20	0x2000	0x20000000
greater than or equal	0xA0	0xA000	0xA0000000
less than	0x40	0x4000	0x40000000
less than and equal	0xC0	0xC000	0xC0000000
force to FALSE	0xE0	0xE000	0xE0000000

d is set to 1, if there is at least one element of a found in any of the ranges. Otherwise, d is set to 3.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 113. Vector Compare Ranges Index with Condition Code					
e	**a**	**b**	**c**	**d**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	int *	ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short		ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int		ARCH(11)

vec_cmprg_or_0_idx: Vector Compare Ranges or Zero Index

```
d = vec_cmprg_or_0_idx(a, b, c)
```

Returns the lowest byte-index of the element of a that is 0 or within any of the ranges specified by b and c. Each even-odd element pairs of b define values for the limits of the ranges. The corresponding even-odd pairs of elements in c control how the comparison to be performed, in the following way:

Comparison	**for vector unsigned char**	**for vector unsigned short**	**for vector unsigned int**
ignore - result of comparison always TRUE	0	0	0
equal	0x80	0x8000	0x80000000
not equal	0x60	0x6000	0x60000000
greater than	0x20	0x2000	0x20000000
greater than or equal	0xA0	0xA000	0xA0000000
less than	0x40	0x4000	0x40000000
less than and equal	0xC0	0xC000	0xC0000000
force to FALSE	0xE0	0xE000	0xE0000000

The result is the lowest byte-index from element of a that is 0 or contained in any of the specified ranges. Otherwise, the result is 16.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 114. Vector Compare Ranges or Zero Index				
d	**a**	**b**	**c**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11)

Example:

```
vector unsigned int a = {1, 0, 22, 33};
vector unsigned int b = {10, 20, 30, 40};
vector unsigned int c = {0x20000000, 0x40000000, 0x20000000,
                         0x40000000}; // {GT, LT, GT, LT}

vector unsigned int d = vec_cmprg_or_0_idx(a, b, c); // byte 7 of d = 4
```

In this example, each element of a is tested to be (equals to 0) OR (> 10 AND < 5) OR (>30 AND < 40), the second element (byte index 4) is the first element satisfying the condition.

vec_cmprg_or_0_idx_cc: Vector Compare Ranges or Zero Index with Condition Code

```
e = vec_cmprg_or_0_idx_cc(a, b, c, d)
```

Returns the lowest byte-index of the element of a that is 0 or within any of the ranges specified by b and c. Each even-odd element pairs of b define values for the limits of the ranges. The corresponding even-odd pairs of elements in c control how the comparison to be performed, in the following way:

Comparison	for vector unsigned char	for vector unsigned short	for vector unsigned int
ignore - result of comparison always TRUE	0	0	0
equal	0x80	0x8000	0x80000000
not equal	0x60	0x6000	0x60000000
greater than	0x20	0x2000	0x20000000
greater than or equal	0xA0	0xA000	0xA0000000
less than	0x40	0x4000	0x40000000
less than and equal	0xC0	0xC000	0xC0000000
force to FALSE	0xE0	0xE000	0xE0000000

d is set to one of the following:

- 0 - if 0 was found on an element of a, before an element was found within the specified range.
- 1 - if no element of a is 0, and there is at least one element of a found in any of the ranges.
- 2 - if 0 was found on an element of a after an element was found within the specified range.
- 3 - no element is 0 and found to be within any of the specified range.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 115. Vector Compare Ranges or Zero Index with Condition Code

e	a	b	c	d	MIN ARCH
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	int *	ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short		ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int		ARCH(11)

Find Any Element

This section describes vector built-in functions for element searching.

vec_find_any_eq: Vector Find Any Element Equal

```
d = vec_find_any_eq(a, b)
```

Find element of a from any element of b with an equal value.

For each element of the result, the value of each bit is 1 if the corresponding elements of a equal any element of b. Otherwise, the value of each bit is 0.

Table 116. Vector Find Any Element Equal

d	a	b	MIN ARCH
vector bool char	vector signed char	vector signed char	ARCH(11)
	vector bool char	vector bool char	ARCH(11)
	vector unsigned char	vector unsigned char	ARCH(11)
vector bool short	vector signed short	vector signed short	ARCH(11)
	vector bool short	vector bool short	ARCH(11)
	vector unsigned short	vector unsigned short	ARCH(11)
vector bool int	vector signed int	vector signed int	ARCH(11)
	vector bool int	vector bool int	ARCH(11)
	vector unsigned int	vector unsigned int	ARCH(11)

Example:

```
vector signed int a = {1, -2, 3, -4};
vector signed int b = {-5, 3, -7, 8};

vector bool int d = vec_find_any_eq(a, b); // d = {0, 0, 0xFFFFFFFF, 0}
```

vec_find_any_eq_cc: Vector Find Any Element Equal with Condition Code

```
d = vec_find_any_eq_cc(a, b, c)
```

Find element of a from any element of b with an equal value.

For each element of the result, the value of each bit is 1 if the corresponding elements of a equal any element of b. Otherwise, the value of each bit is 0. c is set to 1, if there is at least one element of a find a match with b, otherwise c is set to 3.

Table 117. Vector Find Any Element Equal with Condition Code

d	a	b	c	MIN ARCH
vector bool char	vector signed char	vector signed char	int *	ARCH(11)
	vector bool char	vector bool char		ARCH(11)
	vector unsigned char	vector unsigned char		ARCH(11)
vector bool short	vector signed short	vector signed short		ARCH(11)
	vector bool short	vector bool short		ARCH(11)
	vector unsigned short	vector unsigned short		ARCH(11)
vector bool int	vector signed int	vector signed int		ARCH(11)
	vector bool int	vector bool int		ARCH(11)
	vector unsigned int	vector unsigned int		ARCH(11)

vec_find_any_eq_idx: Vector Find Any Element Equal Index

```
d = vec_find_any_eq_idx(a, b)
```

Find the lowest byte-index of element of a from any element of b with an equal value. The result is the lowest byte-index from element of a, if it is found. Otherwise, the result is 16.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 118. Vector Find Any Element Equal Index

d	**a**	**b**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	ARCH(11)
vector unsigned char	vector bool char	vector bool char	ARCH(11)
	vector unsigned char	vector unsigned char	ARCH(11)
vector signed short	vector signed short	vector signed short	ARCH(11)
vector unsigned short	vector bool short	vector bool short	ARCH(11)
	vector unsigned short	vector unsigned short	ARCH(11)
vector signed int	vector signed int	vector signed int	ARCH(11)
vector unsigned int	vector bool int	vector bool int	ARCH(11)
	vector unsigned int	vector unsigned int	ARCH(11)

Example 1:

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {5, 3, 7, 8};

vector unsigned int d = vec_find_any_eq_idx(a,b); // byte 7 of d = 8
```

In this example, the third element (byte index 8) of a was found in the vector b.

Example 2:

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {5, 6, 7, 8};

vector unsigned int d = vec_find_any_eq_idx(a,b); // byte 7 of d = 16
```

In this example, no element from a was found in b, so 16 is returned.

vec_find_any_eq_idx_cc: Vector Find Any Element Equal Index with Condition Code

```
d = vec_find_any_eq_idx_cc(a, b, c)
```

Find the lowest byte-index of element of a from any element of b with an equal value. If it is found, the result is the lowest byte-index from element of a, and c is set to 1. Otherwise, the result is 16, with c set to 3.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 119. Vector Find Any Element Equal Index with Condition Code				
d	**a**	**b**	**c**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	int *	ARCH(11)
vector unsigned char	vector bool char	vector bool char		ARCH(11)
	vector unsigned char	vector unsigned char		ARCH(11)
vector signed short	vector signed short	vector signed short		ARCH(11)
vector unsigned short	vector bool short	vector bool short		ARCH(11)
	vector unsigned short	vector unsigned short		ARCH(11)
vector signed int	vector signed int	vector signed int		ARCH(11)
vector unsigned int	vector bool int	vector bool int		ARCH(11)
	vector unsigned int	vector unsigned int		ARCH(11)

Example 1:

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {5, 3, 7, 8};
int c = 0;

vector unsigned int d = vec_find_any_eq_idx_cc(a,b,&c); // byte 7 of d = 8, c = 1
```

In this example, the third element (byte index 8) of a was found in the vector b.

Example 2:

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {5, 6, 7, 8};
int c = 0;

vector unsigned int d = vec_find_any_eq_idx_cc(a,b,&c); // byte 7 of d = 16, c = 3
```

In this example, the no element from a was found in b, so 16 is returned.

vec_find_any_eq_or_0_idx: Vector Find Any Element Equal or Zero Index

```
d = vec_find_any_eq_or_0_idx(a, b)
```

Find the byte-index of element of a from any element of b with an equal value, or the byte-index of element of a is 0. The result is the lowest byte-index from element of a, if it is found to match those conditions. Otherwise, the result is 16.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 120. Vector Find Any Element Equal or Zero Index			
d	**a**	**b**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	ARCH(11)
vector unsigned char	vector bool char	vector bool char	ARCH(11)
	vector unsigned char	vector unsigned char	ARCH(11)
vector signed short	vector signed short	vector signed short	ARCH(11)
vector unsigned short	vector bool short	vector bool short	ARCH(11)
	vector unsigned short	vector unsigned short	ARCH(11)
vector signed int	vector signed int	vector signed int	ARCH(11)

Table 120. Vector Find Any Element Equal or Zero Index (continued)			
d	**a**	**b**	**MIN ARCH**
vector unsigned int	vector bool int	vector bool int	ARCH(11)
	vector unsigned int	vector unsigned int	ARCH(11)

Example:

```
vector unsigned int a = {1, 2, 0, 4};
vector unsigned int b = {5, 6, 7, 8};

vector unsigned int d = vec_find_any_eq_or_0_idx(a,b); // byte 7 of d = 8
```

In this example, the first and second elements of a are not found in the vector b, and the third element (byte index 8) is a 0.

vec_find_any_eq_or_0_idx_cc: Vector Find Any Element Equal or Zero Index with Condition Code

```
d = vec_find_any_eq_or_0_idx_cc(a, b, c)
```

Find the byte-index of element of a from any element of b with an equal value, or the byte-index of element of a is 0. The result is the lowest byte-index from element of a, if it is found to match those conditions. Otherwise, the result is 16.

c would be set to one of the following values:

- 0 - if no element of a matches any element of b with an equal value, and there is at least one element from a with a value of 0.
- 1 - if at least one element of a matches any element of b with an equal value, and no elements of a with a value of 0.
- 2 - if at least one element of a matches any element of b with an equal value, and there is at least one element from a has a value of 0.
- 3 - if no element of a matches any element of b with an equal value, and there is no element from a with a value of 0.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 121. Vector Find Any Element Equal or Zero Index with Condition Code				
d	**a**	**b**	**c**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	int *	ARCH(11)
vector unsigned char	vector bool char	vector bool char		ARCH(11)
	vector unsigned char	vector unsigned char		ARCH(11)
vector signed short	vector signed short	vector signed short		ARCH(11)
vector unsigned short	vector bool short	vector bool short		ARCH(11)
	vector unsigned short	vector unsigned short		ARCH(11)
vector signed int	vector signed int	vector signed int		ARCH(11)
vector unsigned int	vector bool int	vector bool int		ARCH(11)
	vector unsigned int	vector unsigned int		ARCH(11)

vec_find_any_ne: Vector Find Any Element Not Equal

```
d = vec_find_any_ne(a, b)
```

Find element of a from any element of b with a not equal value.

For each element of the result, the value of each bit is 1 if the corresponding elements of a does not equal to any element of b. Otherwise, the value of each bit is 0.

Table 122. Vector Find Any Element Not Equal

d	a	b	MIN ARCH
vector bool char	vector signed char	vector signed char	ARCH(11)
	vector bool char	vector bool char	ARCH(11)
	vector unsigned char	vector unsigned char	ARCH(11)
vector bool short	vector signed short	vector signed short	ARCH(11)
	vector bool short	vector bool short	ARCH(11)
	vector unsigned short	vector unsigned short	ARCH(11)
vector bool int	vector signed int	vector signed int	ARCH(11)
	vector bool int	vector bool int	ARCH(11)
	vector unsigned int	vector unsigned int	ARCH(11)

Example:

```
vector signed int a = {1, -2, 3, -4};
vector signed int b = {-5, 3, -7, 8};

vector bool int d = vec_find_any_ne(a, b);
// d = {0xFFFFFFFF, 0xFFFFFFFF, 0, 0xFFFFFFFF}
```

vec_find_any_ne_cc: Vector Find Any Element Not Equal with Condition Code

```
d = vec_find_any_ne_cc(a, b, c)
```

Find element of a from any element of b with a not equal value.

For each element of the result, the value of each bit is 1 if the corresponding elements of a does not equal to any element of b. Otherwise, the value of each bit is 0. c is set to 1, if there is at least one element of **a** didn't find a match with b, otherwise c is set to 3.

Table 123. Vector Find Any Element Not Equal with Condition Code

d	a	b	c	MIN ARCH
vector bool char	vector signed char	vector signed char	int *	ARCH(11)
	vector bool char	vector bool char		ARCH(11)
	vector unsigned char	vector unsigned char		ARCH(11)
vector bool short	vector signed short	vector signed short		ARCH(11)
	vector bool short	vector bool short		ARCH(11)
	vector unsigned short	vector unsigned short		ARCH(11)
vector bool int	vector signed int	vector signed int		ARCH(11)
	vector bool int	vector bool int		ARCH(11)
	vector unsigned int	vector unsigned int		ARCH(11)

vec_find_any_ne_idx: Vector Find Any Element Not Equal Index

```
d = vec_find_any_ne_idx(a, b)
```

Find the lowest byte-index of element of a from any element of b with a not equal value. The result is the lowest byte-index from element of a, if it is found. Otherwise, the result is 16.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 124. Vector Find Any Element Not Equal Index

d	a	b	MIN ARCH
vector signed char	vector signed char	vector signed char	ARCH(11)
vector unsigned char	vector bool char	vector bool char	ARCH(11)
	vector unsigned char	vector unsigned char	ARCH(11)
vector signed short	vector signed short	vector signed short	ARCH(11)
vector unsigned short	vector bool short	vector bool short	ARCH(11)
	vector unsigned short	vector unsigned short	ARCH(11)
vector signed int	vector signed int	vector signed int	ARCH(11)
vector unsigned int	vector bool int	vector bool int	ARCH(11)
	vector unsigned int	vector unsigned int	ARCH(11)

Example 1:

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {1, 5, 3, 4};

vector unsigned int d = vec_find_any_ne_idx(a,b); // byte 7 of d = 4
```

In this example, the second element (byte index 4) of a was found to be not equal to any element in b.

Example 2:

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {1, 2, 3, 4};

vector unsigned int d = vec_find_any_ne_idx(a,b); // byte 7 of d = 16
```

In this example, no element from a was found to be not equal to any element in b, so 16 is returned.

vec_find_any_ne_idx_cc: Vector Find Any Element Not Equal Index with Condition Code

```
d = vec_find_any_ne_idx_cc(a, b, c)
```

Find the lowest byte-index of element of a from any element of b with a not equal value. If it is found, the result is the lowest byte-index from element of a, and c is set to 1. Otherwise, the result is 16, with c set to 3.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 125. Vector Find Any Element Not Equal Index with Condition Code				
d	**a**	**b**	**c**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	int *	ARCH(11)
vector unsigned char	vector bool char	vector bool char		ARCH(11)
	vector unsigned char	vector unsigned char		ARCH(11)
vector signed short	vector signed short	vector signed short		ARCH(11)
vector unsigned short	vector bool short	vector bool short		ARCH(11)
	vector unsigned short	vector unsigned short		ARCH(11)
vector signed int	vector signed int	vector signed int		ARCH(11)
vector unsigned int	vector bool int	vector bool int		ARCH(11)
	vector unsigned int	vector unsigned int		ARCH(11)

Example 1:

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {1, 5, 3, 4};
int c = 0;

vector unsigned int d = vec_find_any_ne_idx_cc(a,b,&c); // byte 7 of d = 4, c = 1
```

In this example, the second element (byte index 4) of a was found to be not equal to any element in the vector b.

Example 2:

```
vector unsigned int a = {1, 2, 3, 4};
vector unsigned int b = {1, 2, 3, 4};
int c = 0;

vector unsigned int d = vec_find_any_ne_idx_cc(a,b,&c); // byte 7 of d = 16, c = 3
```

vec_find_any_ne_or_0_idx: Vector Find Any Element Not Equal or Zero Index

```
d = vec_find_any_ne_or_0_idx(a, b)
```

Find the byte-index of element of a from any element of b with a not equal value, or the byte-index of element of a is 0. The result is the lowest byte-index from element of a, if it is found to match those conditions. Otherwise, the result is 16.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 126. Vector Find Any Element Not Equal or Zero Index			
d	**a**	**b**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	ARCH(11)
vector unsigned char	vector bool char	vector bool char	ARCH(11)
	vector unsigned char	vector unsigned char	ARCH(11)
vector signed short	vector signed short	vector signed short	ARCH(11)
vector unsigned short	vector bool short	vector bool short	ARCH(11)
	vector unsigned short	vector unsigned short	ARCH(11)
vector signed int	vector signed int	vector signed int	ARCH(11)

Table 126. Vector Find Any Element Not Equal or Zero Index (continued)			
d	**a**	**b**	**MIN ARCH**
vector unsigned int	vector bool int	vector bool int	ARCH(11)
	vector unsigned int	vector unsigned int	ARCH(11)

Example:

```
vector unsigned int a = {1, 2, 0, 4};
vector unsigned int b = {1, 2, 3, 4};

vector unsigned int d = vec_find_any_ne_or_0_idx(a,b); // byte 7 of d = 8
```

In this example, the first and second elements of a are found in the vector b, and the third element (byte index 8) is a 0.

vec_find_any_ne_or_0_idx_cc: Vector Find Any Element Not Equal or Zero Index with Condition Code

```
d = vec_find_any_ne_or_0_idx_cc(a, b, c)
```

Find the byte-index of element of a from any element of b with a not equal value, or the byte-index of element of a is 0. The result is the lowest byte-index from element of a, if it is found to match those conditions. Otherwise, the result is 16.

c would be set to one of the following values:

- 0 - if no element of a matches any element of b with a not equal value, and there is at least one element from a with a value of 0.
- 1 - if at least one element of a matches any element of b with a not equal value, and no elements of a with a value of 0.
- 2 - if at least one element of a matches any element of b with a not equal value, and there is at least one element from a has a value of 0.
- 3 - if no element of a matches any element of b with a not equal value, and there is no element from a with a value of 0.

The result is placed into byte element seven of the returned vector, and all other bytes are set to 0.

Table 127. Vector Find Any Element Not Equal or Zero Index with Condition Code				
d	**a**	**b**	**c**	**MIN ARCH**
vector signed char	vector signed char	vector signed char	int *	ARCH(11)
vector unsigned char	vector bool char	vector bool char		ARCH(11)
	vector unsigned char	vector unsigned char		ARCH(11)
vector signed short	vector signed short	vector signed short		ARCH(11)
vector unsigned short	vector bool short	vector bool short		ARCH(11)
	vector unsigned short	vector unsigned short		ARCH(11)
vector signed int	vector signed int	vector signed int		ARCH(11)
vector unsigned int	vector bool int	vector bool int		ARCH(11)
	vector unsigned int	vector unsigned int		ARCH(11)

Gather and Scatter

This section describes vector built-in functions for gathering and scattering elements.

vec_bperm_u128: Vector Bit Permute

```
d = vec_bperm_u128(a, b)
```

Gathers up to 16 1-bit values from a quadword in the specified order, and places them in the specified order in bits 48 - 63 of the result vector register, with the rest of the result zeroed.

For each `i` (0 <= `i` < 16), suppose `index` denote the byte value of the `i`-th element of b.

If `index` is greater than or equal to 128, bit 48+`i` of the result is set to 0.

If `index` is smaller than 128, bit 48+`i` of the result is set to the value of the `index`-th bit of input a.

All other bits are set to 0.

For example:

```
vector unsigned char a = (vector unsigned char) (65,1,1,1,1,1,1,1, 1,1,1,1,1,1,1,1);
vector unsigned char b = (vector unsigned char)
(0,0,0,0,1,1,1,1,128,128,128,128,255,255,255,255);
vector unsigned long long d = vec_bperm_u128(a, b);       //d[0]=0xF00, d[1]=0
```

Table 128. Vector Bit Permute

d	a	b	MIN ARCH
vector unsigned long long	vector unsigned char	vector unsigned char	ARCH(12)

vec_extract: Vector Extract

```
d = vec_extract(a, b)
```

Returns the value of element b from the vector a. This function uses the modulo arithmetic on b to determine the element number. For example, if b is out of range, the compiler uses b modulo the number of elements in the vector to determine the element position.

Table 129. Vector Extract

d	a	b	MIN ARCH
signed char	vector signed char	signed int	ARCH(11) 1
unsigned char	vector bool char		ARCH(11) 1
	vector unsigned char		ARCH(11) 1
signed short	vector signed short		ARCH(11) 1
unsigned short	vector bool short		ARCH(11) 1
	vector unsigned short		ARCH(11) 1
signed int	vector signed int		ARCH(11) 1
unsigned int	vector bool int		ARCH(11) 1
	vector unsigned int		ARCH(11) 1
signed long long	vector signed long long		ARCH(11) 1
unsigned long long	vector bool long long		ARCH(11) 1
	vector unsigned long long		ARCH(11) 1
float	vector float		ARCH(12) 1
double	vector double		ARCH(11) 1

Table 129. Vector Extract (continued)			
d	**a**	**b**	**MIN ARCH**
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_gather_element: Vector Gather Element

```
e = vec_gather_element(a, b, c, d)
```

Returns a copy of the vector a with the value of its element d replaced by `*(c+b[d])`.

Table 130. Vector Gather Element					
e	**a**	**b**	**c**	**d**	**MIN ARCH**
vector signed int	vector signed int	vector unsigned int	const signed int *	0-3	ARCH(11)
vector bool int	vector bool int		const unsigned int *		ARCH(11)
vector unsigned int	vector unsigned int				ARCH(11)
vector signed long long	vector signed long long	vector unsigned long long	const signed long long *	0-1	ARCH(11)
vector bool long long	vector bool long long		const unsigned long long *		ARCH(11)
vector unsigned long long	vector unsigned long long				ARCH(11)
vector float	vector float	vector unsigned int	const float *	0-3	ARCH(12)
vector double	vector double	vector unsigned long long	const double *	0-1	ARCH(11)

Example:

```
unsigned int a1[10] = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
unsigned int a2[10] = {20, 21, 22, 23, 24, 25, 26, 27, 28, 29};
vector unsigned int v1= {1, 2, 3, 4}, v2 = {1, 2, 3, 4};
vector unsigned int v3 = {5 * sizeof(int), 8 * sizeof(int),
                          9 * sizeof(int), 6 * sizeof(int)};

v1 = vec_gather_element (v1, v3, a1, 0); // v1 = {15, 2, 3, 4}
v2 = vec_gather_element (v2, v3, a2, 0); // v2 = {25, 2, 3, 4}
```

vec_insert: Vector Insert

```
d = vec_insert(a, b, c)
```

Returns a copy of the vector b with the value of its element c replaced by a. This function uses the modulo arithmetic on c to determine the element number. For example, if c is out of range, the compiler uses c modulo the number of elements in the vector to determine the element position.

Table 131. Vector Insert				
d	**a**	**b**	**c**	**MIN ARCH**
vector signed char	signed char	vector signed char	signed int	ARCH(11) 1
vector unsigned char	unsigned char	vector unsigned char		ARCH(11) 1
vector signed short	signed short	vector signed short		ARCH(11) 1
vector unsigned short	unsigned short	vector unsigned short		ARCH(11) 1
vector signed int	signed int	vector signed int		ARCH(11) 1
vector unsigned int	unsigned int	vector unsigned int		ARCH(11) 1
vector signed long long	signed long long	vector signed long long		ARCH(11) 1
vector unsigned long long	unsigned long long	vector unsigned long long		ARCH(11) 1
vector float	float	vector float		ARCH(12) 1
vector double	double	vector double		ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.				

vec_insert_and_zero: Vector Insert and Zero

```
d = vec_insert_and_zero(a)
```

Returns vector d with the rightmost sub-element or element of the leftmost doubleword element set to what is pointed to by a. The bit positions of all other elements are set to zero.

Table 132. Vector Insert and Zero		
d	**a**	**MIN ARCH**
vector unsigned char	const unsigned char *	ARCH(11)
vector signed char	const signed char *	ARCH(11)
vector unsigned short	const unsigned short *	ARCH(11)
vector signed short	const signed short *	ARCH(11)
vector unsigned int	const unsigned int *	ARCH(11)
vector signed int	const signed int *	ARCH(11)
vector unsigned long long	const unsigned long long *	ARCH(11)
vector signed long long	const signed long long *	ARCH(11)
vector float	const float *	ARCH(12)
vector double	const double *	ARCH(11)

vec_perm: Vector Permute

```
d = vec_perm(a, b, c)
```

Returns a vector that contains some elements of two vectors, in the order specified by a third vector.

Each byte of the result is selected by using the least significant 5 bits of the corresponding byte of c as an index into the concatenated bytes of a and b.

Note: The vector generate mask built-in function `vec_genmask` could help generate the mask c.

Table 133. Vector Permute

d	a	b	c	MIN ARCH
vector signed char	vector signed char	vector signed char	vector unsigned char	ARCH(11) 1
vector bool char	vector bool char	vector bool char		ARCH(11) 1
vector unsigned char	vector unsigned char	vector unsigned char		ARCH(11) 1
vector signed short	vector signed short	vector signed short		ARCH(11) 1
vector bool short	vector bool short	vector bool short		ARCH(11) 1
vector unsigned short	vector unsigned short	vector unsigned short		ARCH(11) 1
vector signed int	vector signed int	vector signed int		ARCH(11) 1
vector bool int	vector bool int	vector bool int		ARCH(11) 1
vector unsigned int	vector unsigned int	vector unsigned int		ARCH(11) 1
vector signed long long	vector signed long long	vector signed long long		ARCH(11) 1
vector bool long long	vector bool long long	vector bool long long		ARCH(11) 1
vector unsigned long long	vector unsigned long long	vector unsigned long long		ARCH(11) 1
vector float	vector float	vector float		ARCH(12) 1
vector double	vector double	vector double		ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_promote: Vector Promote

```
d = vec_promote(a, b)
```

Returns a vector with a in element position b. The result is a vector with a in element position b. This function uses modulo arithmetic on b to determine the element number. For example, if b is out of range, the compiler uses b modulo the number of elements in the vector to determine the element position. The other elements of the vector are undefined.

Table 134. Vector Promote			
d	**a**	**b**	**MIN ARCH**
vector signed char	signed char	signed int	ARCH(11)
vector unsigned char	unsigned char		ARCH(11)
vector signed short	signed short		ARCH(11)
vector unsigned short	unsigned short		ARCH(11)
vector signed int	signed int		ARCH(11)
vector unsigned int	unsigned int		ARCH(11)
vector signed long long	signed long long		ARCH(11)
vector unsigned long long	unsigned long long		ARCH(11)
vector float	float		ARCH(12)
vector double	double		ARCH(11)

vec_scatter_element: Vector Scatter Element

```
vec_scatter_element(a, b, c, d)
```

Store vector element `a[d]` to `*(c+b[d])`.

Table 135. Vector Scatter Element				
a	**b**	**c**	**d**	**MIN ARCH**
vector signed int	vector unsigned int	signed int *	0 - 3	ARCH(11)
vector bool int		unsigned int *		ARCH(11)
vector unsigned int				ARCH(11)
vector signed long long	vector unsigned long long	signed long long *	0 - 1	ARCH(11)
vector bool long long		unsigned long long *		ARCH(11)
vector unsigned long long				ARCH(11)
vector float	vector unsigned int	float *	0-3	ARCH(12)
vector double	vector unsigned long long	double *	0 - 1	ARCH(11)

vec_sel: Vector Select

```
d = vec_sel(a, b, c)
```

Returns a vector containing the value of either a or b depending on the value of c. Each bit of the result vector has the value of the corresponding bit of a if the corresponding bit of c is 0, or the value of the corresponding bit of b otherwise.

Table 136. Vector Select				
d	**a**	**b**	**c**	**MIN ARCH**
vector bool char	vector bool char	vector bool char	vector bool char	ARCH(11) 1
			vector unsigned char	ARCH(11) 1

Table 136. Vector Select (continued)

d	a	b	c	MIN ARCH
vector signed char	vector signed char	vector signed char	vector bool char	ARCH(11) 1
			vector unsigned char	ARCH(11) 1
vector unsigned char	vector unsigned char	vector unsigned char	vector bool char	ARCH(11) 1
			vector unsigned char	ARCH(11) 1
vector bool short	vector bool short	vector bool short	vector bool short	ARCH(11) 1
			vector unsigned short	ARCH(11) 1
vector signed short	vector signed short	vector signed short	vector bool shot	ARCH(11) 1
			vector unsigned short	ARCH(11) 1
vector unsigned short	vector unsigned short	vector unsigned short	vector bool short	ARCH(11) 1
			vector unsigned short	ARCH(11) 1
vector bool int	vector bool int	vector bool int	vector bool int	ARCH(11) 1
			vector unsigned int	ARCH(11) 1
vector signed int	vector signed int	vector signed int	vector bool int	ARCH(11) 1
			vector unsigned int	ARCH(11) 1
vector unsigned int	vector unsigned int	vector unsigned int	vector bool int	ARCH(11) 1
			vector unsigned int	ARCH(11) 1
vector bool long long	vector bool long long	vector bool long long	vector bool long long	ARCH(11) 1
			vector unsigned long long	ARCH(11) 1
vector signed long long	vector signed long long	vector signed long long	vector bool long long	ARCH(11) 1
			vector unsigned long long	ARCH(11) 1
vector unsigned long long	vector unsigned long long	vector unsigned long long	vector bool long long	ARCH(11) 1
			vector unsigned long long	ARCH(11) 1
vector float	vector float	vector float	vector bool int	ARCH(12) 1
			vector unsigned int	ARCH(12) 1
vector double	vector double	vector double	vector bool long long	ARCH(11) 1
			vector unsigned long long	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

Examples

```
vector signed int a = {1, 2, 3, 4};
vector signed int b = {5, 6, 7, 8};

vector unsigned int e = {9, 10, 11, 12};
vector unsigned int f = {9, 9, 11, 11};

vector bool int c = vec_cmpeq(e, f);          // c = {0xFFFFFFFF, 0, 0xFFFFFFFF, 0}
vector signed int d = vec_sel (a, b, c);      // d = {5, 2, 7, 4}
```

Generate Mask

This section describes vector built-in functions for generating mask.

vec_genmask: Vector Generate Byte Mask

```
d = vec_genmask(a)
```

Generates byte masks for elements in the return vector. For each bit in a, if the bit is one, all bit positions in the corresponding byte element of d are set to ones. Otherwise, if the bit is zero, the corresponding byte element is set to zero.

Table 137. Vector Generate Byte Mask

d	**a**	**MIN ARCH**
vector unsigned char	unsigned short literal	ARCH(11)

vec_genmasks_8: Vector Generate Mask (Byte)

```
d = vec_genmasks_8(a, b)
```

Generates bit masks for elements in the return vector. For each byte element in d, a bit mask is generated. The mask consists of bits set to one starting at the bit position specified by a and ending with the bit position specified by b. All other bit positions are set to zero.

Notes:

- If a or b is greater than 8, the operation is performed as if the value gets modulo by 8.
- If a is greater than b, the operation is perform as if b equals 7.

Table 138. Vector Generate Mask (Byte)

d	**a**	**b**	**MIN ARCH**
vector unsigned char	0 - 255	0 - 255	ARCH(11)

vec_genmasks_16: Vector Generate Mask (Halfword)

```
d = vec_genmasks_16(a, b)
```

Generates bit masks for elements in the return vector. For each halfword element in d, a bit mask is generated. The mask consists of bits set to one starting at the bit position specified by a and ending with the bit position specified by b. All other bit positions are set to zero.

Notes:

- If a or b is greater than 16, the operation is performed as if the value gets modulo by 16.
- If a is greater than b, the operation is perform as if b equals 15.

Table 139. Vector Generate Mask (Halfword)			
d	**a**	**b**	**MIN ARCH**
vector unsigned short	0 - 255	0 - 255	ARCH(11)

vec_genmasks_32: Vector Generate Mask (Word)

```
d = vec_genmasks_32(a, b)
```

Generates bit masks for elements in the return vector. For each element in d, a bit mask is generated. The mask consists of bits set to one starting at the bit position specified by a and ending with the bit position specified by b. All other bit positions are set to zero.

Notes:

- If a or b is greater than 32, the operation is performed as if the value gets modulo by 32.
- If a is greater than b, the operation is perform as if b equals 31.

Table 140. Vector Generate Mask (Word)			
d	**a**	**b**	**MIN ARCH**
vector unsigned int	0 - 255	0 - 255	ARCH(11)

vec_genmasks_64: Vector Generate Mask (Doubleword)

```
d = vec_genmasks_64(a, b)
```

Generates bit masks for elements in the return vector. For each doubleword element in d, a bit mask is generated. The mask consists of bits set to one starting at the bit position specified by a and ending with the bit position specified by b. All other bit positions are set to zero.

Notes:

- If a or b is greater than 64, the operation is performed as if the value gets modulo by 64.
- If a is greater than b, the operation is perform as if b equals 63.

Table 141. Vector Generate Mask (Doubleword)			
d	**a**	**b**	**MIN ARCH**
vector unsigned long long	0 - 255	0 - 255	ARCH(11)

Copy until Zero

This section describes vector built-in functions for copying until a zero is encountered.

vec_cp_until_zero: Vector Copy Until Zero

```
d = vec_cp_until_zero(a)
```

Copies the vector elements from a to d, starting from vector element 0, until the vector element from a contains a value of 0, or the entire vector is copied. The remaining vector elements in d are set to 0.

Table 142. Vector Copy Until Zero		
d	**a**	**MIN ARCH**
vector signed char	vector signed char	ARCH(11)
vector bool char	vector bool char	ARCH(11)
vector unsigned char	vector unsigned char	ARCH(11)

Table 142. Vector Copy Until Zero (continued)		
d	**a**	**MIN ARCH**
vector signed short	vector signed short	ARCH(11)
vector bool short	vector bool short	ARCH(11)
vector unsigned short	vector unsigned short	ARCH(11)
vector signed int	vector signed int	ARCH(11)
vector bool int	vector bool int	ARCH(11)
vector unsigned int	vector unsigned int	ARCH(11)

vec_cp_until_zero_cc: Vector Copy Until Zero

```
d = vec_cp_until_zero_cc(a, b)
```

Copies the vector elements from a to d, starting from vector element 0, until the vector element from a contains a value of 0, or the entire vector is copied. The remaining vector elements in d are set to 0.

c is set to 0, if the entire vector was not copied, due to an element from a contains a value of 0. Otherwise, if all elements of a are non-zero, c is set to 3.

Table 143. Vector Copy Until Zero			
d	**a**	**b**	**MIN ARCH**
vector signed char	vector signed char	int *	ARCH(11)
vector bool char	vector bool char		ARCH(11)
vector unsigned char	vector unsigned char		ARCH(11)
vector signed short	vector signed short		ARCH(11)
vector bool short	vector bool short		ARCH(11)
vector unsigned short	vector unsigned short		ARCH(11)
vector signed int	vector signed int		ARCH(11)
vector bool int	vector bool int		ARCH(11)
vector unsigned int	vector unsigned int		ARCH(11)

Load and Store

This section describes vector built-in functions for loading and storing vectors.

vec_load_bndry: Vector Load to Block Boundary

```
d = vec_load_bndry(a, b)
```

Returns a vector with content loaded from *a, filling the vector starting at byte 0, up to 16 bytes or the byte boundary specified by b. When a boundary condition is encountered, the rest of the bytes in the resulting vector are undefined. The `__lcbb()` built-in function can be used to determine the number of bytes loaded.

Table 144. Vector Load to Block Boundary			
d	**a**	**b**	**MIN ARCH**
vector signed char	const signed char *	64, 128, 256, 512, 1024, 2048, or 4096	ARCH(11)
vector unsigned char	const unsigned char *		ARCH(11)
vector signed short	const signed short *		ARCH(11)
vector unsigned short	const unsigned short *		ARCH(11)
vector signed int	const signed int *		ARCH(11)
vector unsigned int	const unsigned int *		ARCH(11)
vector signed long long	const signed long long *		ARCH(11)
vector unsigned long long	const unsigned long long *		ARCH(11)
vector float	const float *		ARCH(12)
vector double	const double *		ARCH(11)

vec_load_len: Vector Load with Length

```
d = vec_load_len(a, b)
```

Returns a vector with content loaded from *a, filling the vector starting at byte 0, up to the number of bytes specified by b+1. When b is less than 15, the remaining bytes of the returned vector are set to zero. When b is greater than 15, only 16 bytes are loaded.

Table 145. Vector Load with Length			
d	**a**	**b**	**MIN ARCH**
vector signed char	const signed char *	unsigned int	ARCH(11)
vector unsigned char	const unsigned char *		ARCH(11)
vector signed short	const signed short *		ARCH(11)
vector unsigned short	const unsigned short *		ARCH(11)
vector signed int	const signed int *		ARCH(11)
vector unsigned int	const unsigned int *		ARCH(11)
vector signed long long	const signed long long *		ARCH(11)
vector unsigned long long	const unsigned long long *		ARCH(11)
vector float	const float *		ARCH(12)
vector double	const double *		ARCH(11)

vec_load_len_r: Vector Load Rightmost with Length

```
d = vec_load_len_r(a, b)
```

Returns a vector that contains b+1 bytes that are loaded from *a, right justified with the first byte to the left and the last to the right. When b is less than 15, the remaining bytes of the returned vector are set to zero. When b is greater than 15, only 16 bytes are loaded.

Table 146. Vector Load Rightmost with Length			
d	**a**	**b**	**MIN ARCH**
vector unsigned char	const unsigned char *	unsigned int	ARCH(12)

vec_load_pair: Vector Load Pair

```
d = vec_load_pair(a, b)
```

Returns a vector with a on the 0-indexed element, and b on the 1-indexed element.

Note: This function might be emulated.

Table 147. Vector Load Pair			
d	**a**	**b**	**MIN ARCH**
vector signed long long	signed long long	signed long long	ARCH(11)
vector unsigned long long	unsigned long long	unsigned long long	ARCH(11)

vec_store_len: Vector Store with Length

```
d = vec_store_len(a, b, c)
```

Store c+1 number of bytes to *b from the vector a.

Note: If c is greater than 15, only 16 bytes will be stored.

Table 148. Vector Store with Length				
d	**a**	**b**	**c**	**MIN ARCH**
void	vector signed char	signed char *	unsigned int	ARCH(11)
	vector unsigned char	unsigned char *		ARCH(11)
	vector signed short	signed short *		ARCH(11)
	vector unsigned short	unsigned short *		ARCH(11)
	vector signed int	signed int *		ARCH(11)
	vector unsigned int	unsigned int *		ARCH(11)
	vector signed long long	signed long long *		ARCH(11)
	vector unsigned long long	unsigned long long *		ARCH(11)
	vector float	float *		ARCH(12)
	vector double	double *		ARCH(11)

vec_store_len_r: Vector Store Rightmost with Length

```
d = vec_store_len_r(a, b, c)
```

Store c+1 bytes to *b from the right-justified vector a.

Note: If c is greater than 15, only 16 bytes will be stored.

Table 149. Vector Store Rightmost with Length				
d	**a**	**b**	**c**	**MIN ARCH**
void	vector unsigned char	unsigned char *	unsigned int	ARCH(12)

vec_xl: Vector Load

```
d = vec_xl(a, b)
```

Loads a 16-byte vector from the memory address that is specified by the displacement a and the pointer b. This function adds the displacement and the pointer R-value to obtain the address for the load operation.

Note: It is preferred that you use pointers and the indirection operator * instead of this function to load vectors.

Table 150. Vector Load			
d	**a**	**b**	**MIN ARCH**
vector signed char	long	signed char *	ARCH(11)
vector unsigned char		unsigned char *	ARCH(11)
vector signed short		signed short *	ARCH(11)
vector unsigned short		unsigned short *	ARCH(11)
vector signed int		signed int *	ARCH(11)
vector unsigned int		unsigned int *	ARCH(11)
vector signed long long		signed long long *	ARCH(11)
vector unsigned long long		unsigned long long *	ARCH(11)
vector float		float *	ARCH(12)
vector double		double *	ARCH(11)

Note: `vec_xl_be` is available on the XL C/C++ compilers for some other platforms. You can define the following macro to migrate programs from other platforms to the Enterprise Metal C for z/OS compiler.

```
#define vec_xl_be(a, b) vec_xl(a,b)
```

vec_xst: Vector Store

```
d = vec_xst(a, b, c)
```

Stores the elements of the 16-byte vector a to the effective address that is obtained by adding the displacement b in the address c.

Note: It is preferred that you use pointers and the indirection operator * instead of this function to store vectors.

Table 151. Vector Store				
d	**a**	**b**	**c**	**MIN ARCH**
void	vector signed char	long	signed char *	ARCH(11)
	vector unsigned char		unsigned char *	ARCH(11)
	vector signed short		signed short *	ARCH(11)
	vector unsigned short		unsigned short *	ARCH(11)
	vector signed int		signed int *	ARCH(11)
	vector unsigned int		unsigned int *	ARCH(11)
	vector signed long long		signed long long *	ARCH(11)
	vector unsigned long long		unsigned long long *	ARCH(11)
	vector float		float*	ARCH(12)
	vector double		double *	ARCH(11)

Note: `vec_xst_be` is available on the XL C/C++ compilers for some other platforms. You can define the following macro to migrate programs from other platforms to the Enterprise Metal C for z/OS compiler.

```
#define vec_xst_be(a, b) vec_xst(a,b)
```

Logical

This section describes vector built-in functions for logical calculation.

vec_andc: Vector AND With Complement

```
d = vec_andc(a, b)
```

Returns the bitwise AND of the first argument a with the bitwise complement of the second argument b.

Table 152. Vector AND With Complement			
d	**a**	**b**	**MIN ARCH**
vector bool char	vector bool char	vector bool char	ARCH(11) 1
vector signed char	vector signed char	vector signed char	ARCH(11) 1
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11) 1
vector bool short	vector bool short	vector bool short	ARCH(11) 1
vector signed short	vector signed short	vector signed short	ARCH(11) 1
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11) 1
vector bool int	vector bool int	vector bool int	ARCH(11) 1
vector signed int	vector signed int	vector signed int	ARCH(11) 1
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11) 1
vector bool long long	vector bool long long	vector bool long long	ARCH(11) 1
vector signed long long	vector signed long long	vector signed long long	ARCH(11) 1
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(11) 1
vector float	vector float	vector float	ARCH(12) 1

Table 152. Vector AND With Complement (continued)			
d	**a**	**b**	**MIN ARCH**
vector double	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_cntlz: Vector Count Leading Zeros

```
d = vec_cntlz(a)
```

Computes the count of leading zero bits of each element of the input.

Each element of the result is set to the number of leading zeros of the corresponding element of a.

Table 153. Vector Count Leading Zeros		
d	**a**	**MIN ARCH**
vector unsigned char	vector unsigned char	ARCH(11) 1
	vector signed char	ARCH(11) 2
vector unsigned short	vector unsigned short	ARCH(11) 1
	vector signed short	ARCH(11) 2
vector unsigned int	vector unsigned int	ARCH(11) 1
	vector signed int	ARCH(11) 2
vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector signed long long	ARCH(11) 2
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4. 2. This prototype has slightly different semantics than that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4. The return type signedness on d is different from that in OpenPower ABI.		

vec_cnttz: Vector Count Trailing Zeros

```
d = vec_cnttz(a)
```

Computes the count of trailing zero bits of each element of the input.

Each element of the result is set to the number of tailing zeros of the corresponding element of a.

Table 154. Vector Count Trailing Zeros		
d	**a**	**MIN ARCH**
vector unsigned char	vector unsigned char	ARCH(11) 1
	vector signed char	ARCH(11) 2

Table 154. Vector Count Trailing Zeros (continued)		
d	**a**	**MIN ARCH**
vector unsigned short	vector unsigned short	ARCH(11) 1
	vector signed short	ARCH(11) 2
vector unsigned int	vector unsigned int	ARCH(11) 1
	vector signed int	ARCH(11) 2
vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector signed long long	ARCH(11) 2

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

2. This prototype has slightly different semantics than that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4. The return type signedness on d is different from that in OpenPower ABI.

vec_eqv: Vector XNOR

```
d = vec_eqv(a, b)
```

Performs a bitwise XNOR of the given vectors a and b.

Note: This function will not cause IEEE exception on `vector float` and `vector double`.

Table 155. Vector Not Exclusive Or			
d	**a**	**b**	**MIN ARCH**
vector bool char	vector bool char	vector bool char	ARCH(12) 1
vector signed char	vector signed char	vector signed char	ARCH(12) 1
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(12) 1
vector bool short	vector bool short	vector bool short	ARCH(12) 1
vector signed short	vector signed short	vector signed short	ARCH(12) 1
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(12) 1
vector bool int	vector bool int	vector bool int	ARCH(12) 1
vector signed int	vector signed int	vector signed int	ARCH(12) 1
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(12) 1
vector bool long long	vector bool long long	vector bool long long	ARCH(12) 1
vector signed long long	vector signed long long	vector signed long long	ARCH(12) 1
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(12) 1
vector float	vector float	vector float	ARCH(12) 1
vector double	vector double	vector double	ARCH(12) 1

Table 155. Vector Not Exclusive Or (continued)			
d	**a**	**b**	**MIN ARCH**
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_nand: Vector NAND

```
d = vec_nand(a, b)
```

Performs a bitwise NAND of the given vectors a and b.

Note: This function will not cause IEEE exception on `vector float` and `vector double`.

Table 156. Vector NAND			
d	**a**	**b**	**MIN ARCH**
vector bool char	vector bool char	vector bool char	ARCH(12) 1
vector signed char	vector signed char	vector signed char	ARCH(12) 1
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(12) 1
vector bool short	vector bool short	vector bool short	ARCH(12) 1
vector signed short	vector signed short	vector signed short	ARCH(12) 1
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(12) 1
vector bool int	vector bool int	vector bool int	ARCH(12) 1
vector signed int	vector signed int	vector signed int	ARCH(12) 1
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(12) 1
vector bool long long	vector bool long long	vector bool long long	ARCH(12) 1
vector signed long long	vector signed long long	vector signed long long	ARCH(12) 1
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(12) 1
vector float	vector float	vector float	ARCH(12) 1
vector double	vector double	vector double	ARCH(12) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_nor: Vector NOR

```
d = vec_nor(a, b)
```

Performs a bitwise NOR of the given vectors a and b.

Note: This function will not cause IEEE exception on `vector float` and `vector double`.

Table 157. Vector NOR			
d	**a**	**b**	**MIN ARCH**
vector bool char	vector bool char	vector bool char	ARCH(11) 1
vector signed char	vector signed char	vector signed char	ARCH(11) 1
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11) 1
vector bool short	vector bool short	vector bool short	ARCH(11) 1
vector signed short	vector signed short	vector signed short	ARCH(11) 1
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11) 1
vector bool int	vector bool int	vector bool int	ARCH(11) 1
vector signed int	vector signed int	vector signed int	ARCH(11) 1
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11) 1
vector bool long long	vector bool long long	vector bool long long	ARCH(11) 1
vector signed long long	vector signed long long	vector signed long long	ARCH(11) 1
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(11) 1
vector float	vector float	vector float	ARCH(12) 1
vector double	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_orc: Vector OR with Complement

```
d = vec_vec_orc(a, b)
```

Performs a bitwise OR of the vector a with the negated vector b.

Note: This function will not cause IEEE exception on `vector float` and `vector double`.

Table 158. Vector OR with Complement			
d	**a**	**b**	**MIN ARCH**
vector bool char	vector bool char	vector bool char	ARCH(12) 1
vector signed char	vector signed char	vector signed char	ARCH(12) 1
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(12) 1
vector bool short	vector bool short	vector bool short	ARCH(12) 1
vector signed short	vector signed short	vector signed short	ARCH(12) 1
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(12) 1
vector bool int	vector bool int	vector bool int	ARCH(12) 1
vector signed int	vector signed int	vector signed int	ARCH(12) 1
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(12) 1
vector bool long long	vector bool long long	vector bool long long	ARCH(12) 1

Table 158. Vector OR with Complement (continued)			
d	**a**	**b**	**MIN ARCH**
vector signed long long	vector signed long long	vector signed long long	ARCH(12) 1
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(12) 1
vector float	vector float	vector float	ARCH(12) 1
vector double	vector double	vector double	ARCH(12) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_popcnt: Vector Population Count

```
d = vec_popcnt(a)
```

Computes the population count (number of set bits) in each element of the input.

Each element of the result is set to the number of set bits in the corresponding element of the input.

Note: This function emulates the operation, except for `vector signed char` and `vector unsigned char`.

Table 159. Vector Population Count		
d	**a**	**MIN ARCH**
vector unsigned char	vector signed char	ARCH(11) 1
	vector unsigned char	ARCH(11) 1
vector unsigned short	vector signed short	ARCH(11) 1
	vector unsigned short	ARCH(11) 1
vector unsigned int	vector signed int	ARCH(11) 1
	vector unsigned int	ARCH(11) 1
vector unsigned long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

Merge

This section describes vector built-in functions for merging vectors.

vec_mergeh: Vector Merge High

```
d = vec_mergeh(a, b)
```

Merges the most significant ("high") halves of two vectors.

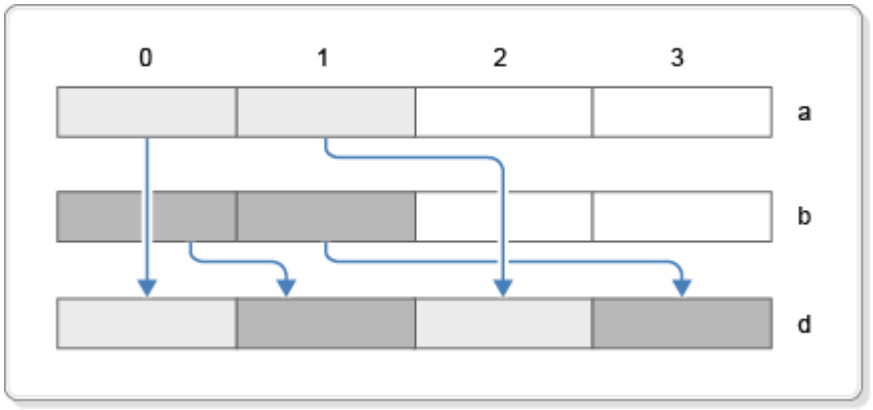


Figure 16. Merge 2 high-order elements (32-bit)

Assume that the elements of each vector are numbered beginning with 0. The even-numbered elements of the result are taken, in order, from the elements in the most significant half of a. The odd-numbered elements of the result are taken, in order, from the elements in the most significant half of b.

Table 160. Vector Merge High

d	a	b	MIN ARCH
vector bool char	vector bool char	vector bool char	ARCH(11) 1
vector signed char	vector signed char	vector signed char	ARCH(11) 1
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11) 1
vector bool short	vector bool short	vector bool short	ARCH(11) 1
vector signed short	vector signed short	vector signed short	ARCH(11) 1
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11) 1
vector bool int	vector bool int	vector bool int	ARCH(11) 1
vector signed int	vector signed int	vector signed int	ARCH(11) 1
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11) 1
vector bool long long	vector bool long long	vector bool long long	ARCH(11) 1
vector signed long long	vector signed long long	vector signed long long	ARCH(11) 1
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(11) 1
vector float	vector float	vector float	ARCH(12) 1
vector double	vector double	vector double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_mergel: Vector Merge Low

```
d = vec_mergel(a, b)
```

Merges the least significant ("low") halves of two vectors.

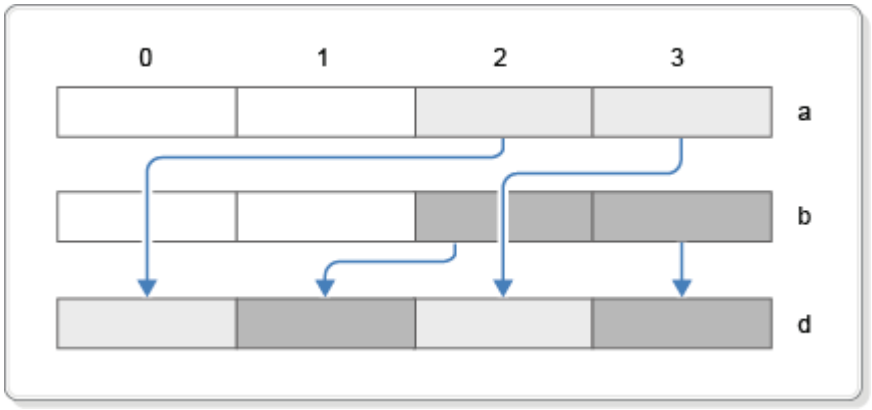


Figure 17. Merge 2 low-order elements (32-bit)

Assume that the elements of each vector are numbered beginning with 0. The even-numbered elements of the result are taken, in order, from the elements in the least significant half of a. The odd-numbered elements of the result are taken, in order, from the elements in the least significant half of b.

Table 161. Vector Merge Low

d	a	b	MIN ARCH
vector bool char	vector bool char	vector bool char	ARCH(11) 1
vector signed char	vector signed char	vector signed char	ARCH(11) 1
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11) 1
vector bool short	vector bool short	vector bool short	ARCH(11) 1
vector signed short	vector signed short	vector signed short	ARCH(11) 1
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11) 1
vector bool int	vector bool int	vector bool int	ARCH(11) 1
vector signed int	vector signed int	vector signed int	ARCH(11) 1
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11) 1
vector bool long long	vector bool long long	vector bool long long	ARCH(11) 1
vector signed long long	vector signed long long	vector signed long long	ARCH(11) 1
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(11) 1
vector float	vector float	vector float	ARCH(12) 1
vector double	vector double	vector double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

Pack and Unpack

This section describes vector built-in functions for pack and unpack.

vec_pack: Vector Pack

```
d = vec_pack(a, b)
```

The value of each element of the result vector is taken from the low-order half of the corresponding element of the result of concatenating a and b.

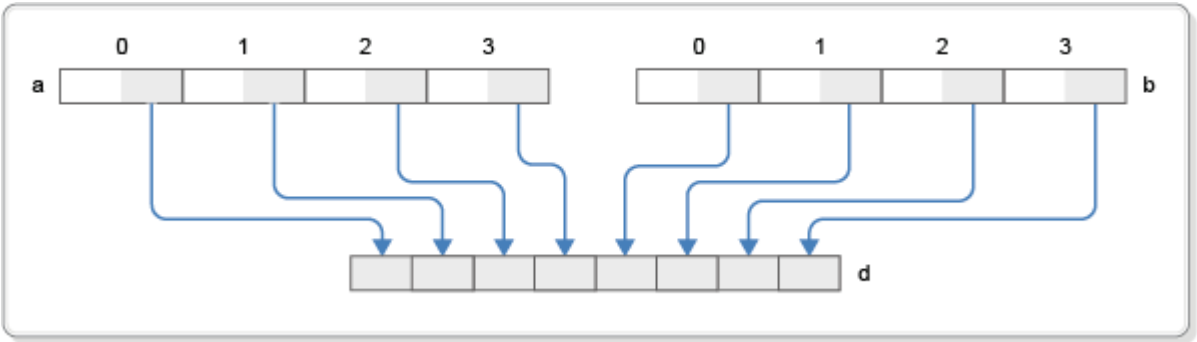


Figure 18. Pack 8 integer elements (32-bit) to 8 integer elements (16-bit)

Table 162. Vector Pack

d	a	b	MIN ARCH
vector signed char	vector signed short	vector signed short	ARCH(11) 1
vector bool char	vector bool short	vector bool short	ARCH(11) 1
vector unsigned char	vector unsigned short	vector unsigned short	ARCH(11) 1
vector signed short	vector signed int	vector signed int	ARCH(11) 1
vector bool short	vector bool int	vector bool int	ARCH(11) 1
vector unsigned short	vector unsigned int	vector unsigned int	ARCH(11) 1
vector signed int	vector signed long long	vector signed long long	ARCH(11) 1
vector bool int	vector bool long long	vector bool long long	ARCH(11) 1
vector unsigned int	vector unsigned long long	vector unsigned long long	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_packs: Vector Pack Saturate

```
d = vec_packs(a, b)
```

The value of each element of the result vector is the saturated value of the corresponding element of the result of concatenating a and b.

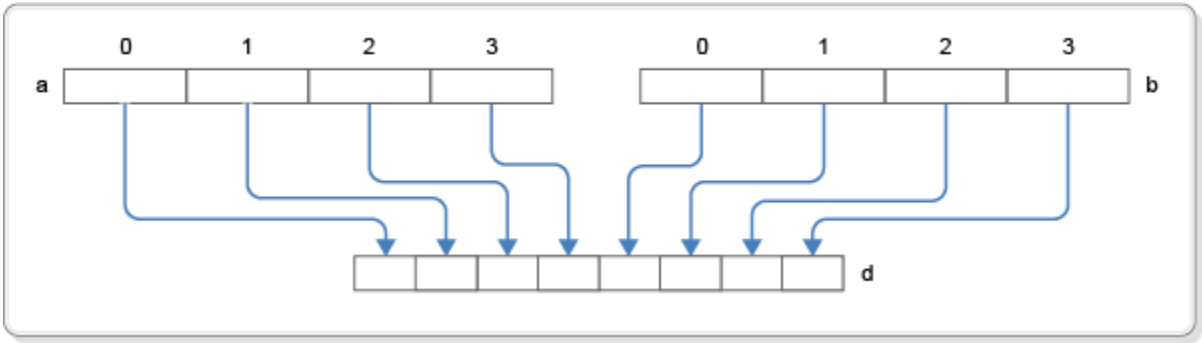


Figure 19. Pack 8 integer elements (32-bit) to 8 integer elements (16-bit)

Table 163. Vector Pack Saturate			
d	**a**	**b**	**MIN ARCH**
vector signed char	vector signed short	vector signed short	ARCH(11) 1
vector unsigned char	vector unsigned short	vector unsigned short	ARCH(11) 1
vector signed short	vector signed int	vector signed int	ARCH(11) 1
vector unsigned short	vector unsigned int	vector unsigned int	ARCH(11) 1
vector signed int	vector signed long long	vector signed long long	ARCH(11) 1
vector unsigned int	vector unsigned long long	vector unsigned long long	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_packs_cc: Vector Pack Saturate Condition Code

```
d = vec_packs_cc(a, b, c)
```

The value of each element of the result vector is the saturated value of the corresponding element of the result of concatenating a and b. The resulting condition code is returned through parameter c. For the signed types, the resulting condition code is from the VECTOR PACK SATURATE (VPKS) instruction. For the unsigned types, the resulting condition code is from the VECTOR PACK LOGICAL SATURATE (VPKLS) instruction.

Table 164. Vector Pack Saturate Condition Code				
d	**a**	**b**	**c**	**MIN ARCH**
vector signed char	vector signed short	vector signed short	int *	ARCH(11)
vector unsigned char	vector unsigned short	vector unsigned short		ARCH(11)
vector signed short	vector signed int	vector signed int		ARCH(11)
vector unsigned short	vector unsigned int	vector unsigned int		ARCH(11)
vector signed int	vector signed long long	vector signed long long		ARCH(11)
vector unsigned int	vector unsigned long long	vector unsigned long long		ARCH(11)

vec_packsu: Vector Pack Saturated Unsigned

```
d = vec_packsu(a, b)
```

The value of each element of the result vector is the saturated unsigned value of the corresponding element of the result of concatenating a and b.

Table 165. Vector Pack Saturated Unsigned			
d	**a**	**b**	**MIN ARCH**
vector unsigned char	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
vector unsigned short	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1

Table 165. Vector Pack Saturated Unsigned (continued)			
d	**a**	**b**	**MIN ARCH**
vector unsigned int	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_packsu_cc: Vector Pack Saturated Unsigned Condition Code

```
d = vec_packsu_cc(a, b, c)
```

The value of each element of the result vector is the saturated unsigned value of the corresponding element of the result of concatenating a and b. The resulting condition code from the VECTOR PACK LOGICAL SATURATE (VPKLS) instruction is returned through parameter c.

Table 166. Vector Pack Saturated Unsigned Condition Code				
d	**a**	**b**	**c**	**MIN ARCH**
vector unsigned char	vector unsigned short	vector unsigned short	int *	ARCH(11)
vector unsigned short	vector unsigned int	vector unsigned int		ARCH(11)
vector unsigned int	vector unsigned long long	vector unsigned long long		ARCH(11)

vec_unpackh: Vector Unpack High Element

```
d = vec_unpackh(a)
```

Unpacks the most significant ("high") half of a vector into a vector with larger elements. The value of each element of the result is the value of the corresponding element of the most significant half of a.

For prototypes with operands a as `vector signed` or `vector bool` types, their resulting values d get sign-extended; while for prototypes with operands a as `vector unsigned` types, their resulting values get 0-extended.

The following diagram illustrates the operation of `vec_unpackh` on the `vector signed` or `vector bool` types.

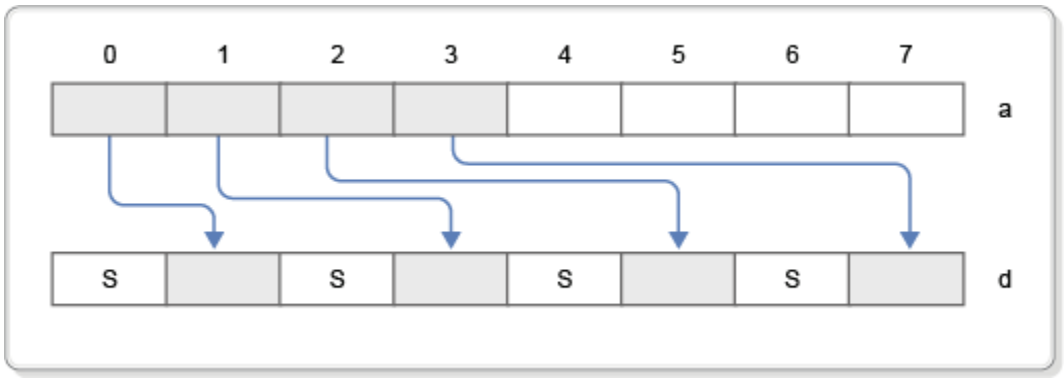


Figure 20. Unpack high-order integer elements (16-bit) to integer elements (32-bit)

Table 167. Vector Unpack High Element		
d	**a**	**MIN ARCH**
vector signed short	vector signed char	ARCH(11) 1
vector bool short	vector bool char	ARCH(11) 1
vector unsigned short	vector unsigned char	ARCH(11)
vector signed int	vector signed short	ARCH(11) 1
vector bool int	vector bool short	ARCH(11) 1
vector unsigned int	vector unsigned short	ARCH(11)
vector signed long long	vector signed int	ARCH(11) 1
vector bool long long	vector bool int	ARCH(11) 1
vector unsigned long long	vector unsigned int	ARCH(11)
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.		

vec_unpackl: Vector Unpack Low Element

```
d = vec_unpackl(a)
```

Unpacks the least significant ("low") half of a vector into a vector with larger elements. The value of each element of the result is the value of the corresponding element of the least significant half of a.

For prototypes with operands a as `vector signed` or `vector bool` types, their resulting values d get sign-extended; while for prototypes with operands a as `vector unsigned` types, their resulting values get 0-extended.

The following diagram illustrates the operation of `vec_unpackl` on the `vector signed` or `vector bool` types.

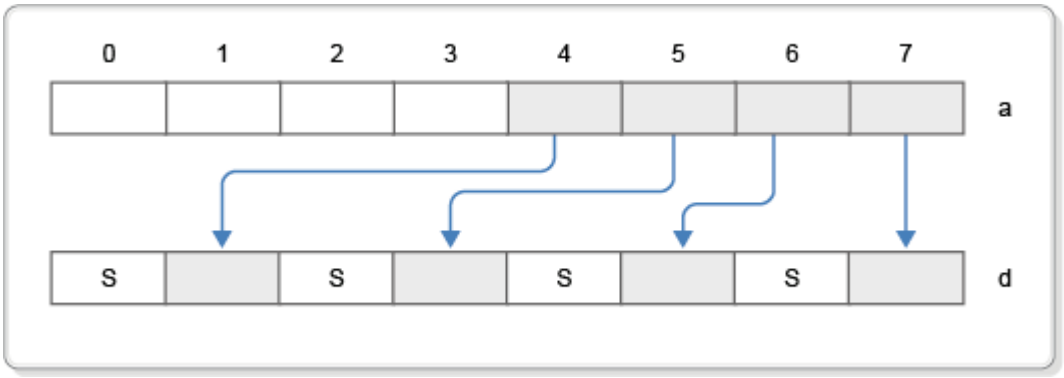


Figure 21. Unpack low-order integer elements (16-bit) to integer elements (32-bit)

Table 168. Vector Unpack Low Element		
d	**a**	**MIN ARCH**
vector signed short	vector signed char	ARCH(11) 1
vector bool short	vector bool char	ARCH(11) 1
vector unsigned short	vector unsigned char	ARCH(11)

Table 168. Vector Unpack Low Element (continued)		
d	**a**	**MIN ARCH**
vector signed int	vector signed short	ARCH(11) 1
vector bool int	vector bool short	ARCH(11) 1
vector unsigned int	vector unsigned short	ARCH(11)
vector signed long long	vector signed int	ARCH(11) 1
vector bool long long	vector bool int	ARCH(11) 1
vector unsigned long long	vector unsigned int	ARCH(11)

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

Replicate

This section describes vector built-in functions for replicating vector elements.

vec_splat: Vector Splat

```
d = vec_splat(a, b)
```

Returns a vector that has all of its elements set to a given value. The value of each element of the result is the value of the element of a specified by b.

Table 169. Vector Splat			
d	**a**	**b**	**MIN ARCH**
vector bool char	vector bool char	0 - 15	ARCH(11)
vector signed char	vector signed char		ARCH(11)
vector unsigned char	vector unsigned char		ARCH(11)
vector bool short	vector bool short	0 - 7	ARCH(11)
vector signed short	vector signed short		ARCH(11)
vector unsigned short	vector unsigned short		ARCH(11)
vector bool int	vector bool int	0 - 3	ARCH(11)
vector signed int	vector signed int		ARCH(11)
vector unsigned int	vector unsigned int		ARCH(11)
vector bool long long	vector bool long long	0 - 1	ARCH(11)
vector signed long long	vector signed long long		ARCH(11)
vector unsigned long long	vector unsigned long long		ARCH(11)
vector float	vector float	0-3	ARCH(12)
vector double	vector double	0-1	ARCH(11)

vec_splat_s8: Vector Splat Signed Byte

```
d = vec_splat_s8(a)
```

Returns a vector with each of the 16 signed 8-bits element equal to the given value.

Table 170. Vector Splat Signed Byte		
d	**a**	**MIN ARCH**
vector signed char	-128 - 127	ARCH(11)

vec_splat_s16: Vector Splat Signed Halfword

```
d = vec_splat_s16(a)
```

Returns a vector with each of the 8 signed 16-bits element equal to the given value.

Table 171. Vector Splat Signed Halfword		
d	**a**	**MIN ARCH**
vector signed short	-2^{15} - 2^{15}-1	ARCH(11)

vec_splat_s32: Vector Splat Signed Word

```
d = vec_splat_s32(a)
```

Returns a vector with each of the 4 signed 32-bits element equal to the given value.

Table 172. Vector Splat Signed Word		
d	**a**	**MIN ARCH**
vector signed int	-2^{15} - 2^{15}-1	ARCH(11)

vec_splat_s64: Vector Splat Signed Doubleword

```
d = vec_splat_s64(a)
```

Returns a vector with each of the 2 signed 64-bits element equal to the given value.

Table 173. Vector Splat Signed Doubleword		
d	**a**	**MIN ARCH**
vector signed long long	-2^{15} - 2^{15}-1	ARCH(11)

vec_splat_u8: Vector Splat Unsigned Byte

```
d = vec_splat_u8(a)
```

Returns a vector with each of the 16 unsigned 8-bits element equal to the given value.

Table 174. Vector Splat Unsigned Byte		
d	**a**	**MIN ARCH**
vector unsigned char	0 - 255	ARCH(11)

vec_splat_u16: Vector Splat Unsigned Halfword

```
d = vec_splat_u16(a)
```

Returns a vector with each of the 8 unsigned 16-bits element equal to the given value.

Table 175. Vector Splat Unsigned Halfword		
d	**a**	**MIN ARCH**
vector unsigned short	0 - 2^{16}-1	ARCH(11)

vec_splat_u32: Vector Splat Unsigned Word

```
d = vec_splat_u32(a)
```

Returns a vector with each of the 4 unsigned 32-bits element equal to the given value.

Table 176. Vector Splat Unsigned Word		
d	**a**	**MIN ARCH**
vector unsigned int	-2^{15} - 2^{15}-1	ARCH(11)

vec_splat_u64: Vector Splat Unsigned Doubleword

```
d = vec_splat_u64(a)
```

Returns a vector with each of the 2 unsigned 64-bits element equal to the given value.

Table 177. Vector Splat Doubleword		
d	**a**	**MIN ARCH**
vector unsigned long long	-2^{15} - 2^{15}-1	ARCH(11)

vec_splats: Vector Splats

```
d = vec_splats(a)
```

Returns a vector of which the value of each element is set to a.

Table 178. Vector Splats		
d	**a**	**MIN ARCH**
vector signed char	signed char	ARCH(11) 1
vector unsigned char	unsigned char	ARCH(11) 1
vector signed short	signed short	ARCH(11) 1
vector unsigned short	unsigned short	ARCH(11) 1
vector signed int	signed int	ARCH(11) 1
vector unsigned int	unsigned int	ARCH(11) 1
vector signed long long	signed long long	ARCH(11) 1
vector unsigned long long	unsigned long long	ARCH(11) 1
vector float	vector float	ARCH(12) 1
vector double	double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

Rotate and Shift

This section describes vector built-in functions for rotate and shift.

vec_rl: Vector Element Rotate Left

```
d = vec_rl(a, b)
```

Rotates each element of a vector left by a given number of bits. Each element of the result is obtained by rotating the corresponding element of a left by the number of bits specified by the corresponding element of b, modulo the number of bits in the element.

Table 179. Vector Element Rotate Left

d	a	b	MIN ARCH
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11) 1
vector signed char	vector signed char		ARCH(11) 1
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11) 1
vector signed short	vector signed short		ARCH(11) 1
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11) 1
vector signed int	vector signed int		ARCH(11) 1
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(11) 1
vector signed long long	vector signed long long		ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_rl_mask: Vector Element Rotate and Insert Under Mask

```
d = vec_rl_mask(a, b, c)
```

Rotates each element of vector a left by a given number of bits c, modulo the number of bits in the element, and overlay with the original vector a depends on the mask b. Each bit of the result is obtained where if the corresponding bit the mask b is 1, it will get the corresponding bit from the intermediate result. Otherwise, if the corresponding bit the mask b is 0, it will get the corresponding bit from a, before the rotation.

Table 180. Vector Element Rotate and Insert Under Mask				
d	**a**	**b**	**c**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	unsigned char literal 0 - 255	ARCH(11)
vector signed char	vector signed char	vector unsigned char		ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short		ARCH(11)
vector signed short	vector signed short	vector unsigned short		ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int		ARCH(11)
vector signed int	vector signed int	vector unsigned int		ARCH(11)
vector unsigned long long	vector unsigned long long	vector unsigned long long		ARCH(11)
vector signed long long	vector signed long long	vector unsigned long long		ARCH(11)

vec_rli: Vector Element Rotate Left Immediate

```
d = vec_rli(a, b)
```

Rotates each element of a vector left by a given number of bits. Each element of the result is obtained by rotating the corresponding element of a left by the number of bits specified by b, modulo the number of bits in the element.

Table 181. Vector Element Rotate Left Immediate			
d	**a**	**b**	**MIN ARCH**
vector unsigned char	vector unsigned char	unsigned long	ARCH(11)
vector signed char	vector signed char		ARCH(11)
vector unsigned short	vector unsigned short		ARCH(11)
vector signed short	vector signed short		ARCH(11)
vector unsigned int	vector unsigned int		ARCH(11)
vector signed int	vector signed int		ARCH(11)
vector unsigned long long	vector unsigned long long		ARCH(11)
vector signed long long	vector signed long long		ARCH(11)

vec_slb: Vector Shift Left by Byte

```
d = vec_slb(a, b)
```

Performs a left shift for a vector by a given number of bytes. Each element of the result is obtained by shifting the corresponding element of a left by the number of bytes specified by bits 1-4 of byte element seven of b. The bits that are shifted out are replaced by zeros.

Table 182. Vector Shift Left by Byte			
d	**a**	**b**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)
		vector signed char	ARCH(11)

Table 182. Vector Shift Left by Byte (continued)			
d	**a**	**b**	**MIN ARCH**
vector signed char	vector signed char	vector unsigned char	ARCH(11)
		vector signed char	ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11)
		vector signed short	ARCH(11)
vector signed short	vector signed short	vector unsigned short	ARCH(11)
		vector signed short	ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11)
		vector signed int	ARCH(11)
vector signed int	vector signed int	vector unsigned int	ARCH(11)
		vector signed int	ARCH(11)
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(11)
		vector signed long long	ARCH(11)
vector signed long long	vector signed long long	vector unsigned long long	ARCH(11)
		vector signed long long	ARCH(11)
vector float	vector float	vector unsigned int	ARCH(12)
		vector signed int	ARCH(12)
vector double	vector double	vector unsigned long long	ARCH(11)
		vector signed long long	ARCH(11)

vec_sld: Vector Shift Left Double by Byte

```
d = vec_sld(a, b, c)
```

Performs a left shift for two concatenated vectors by a given number of bytes. The result is the most significant 16 bytes obtained by concatenating a and b, and shifting left by the number of bytes specified by c.

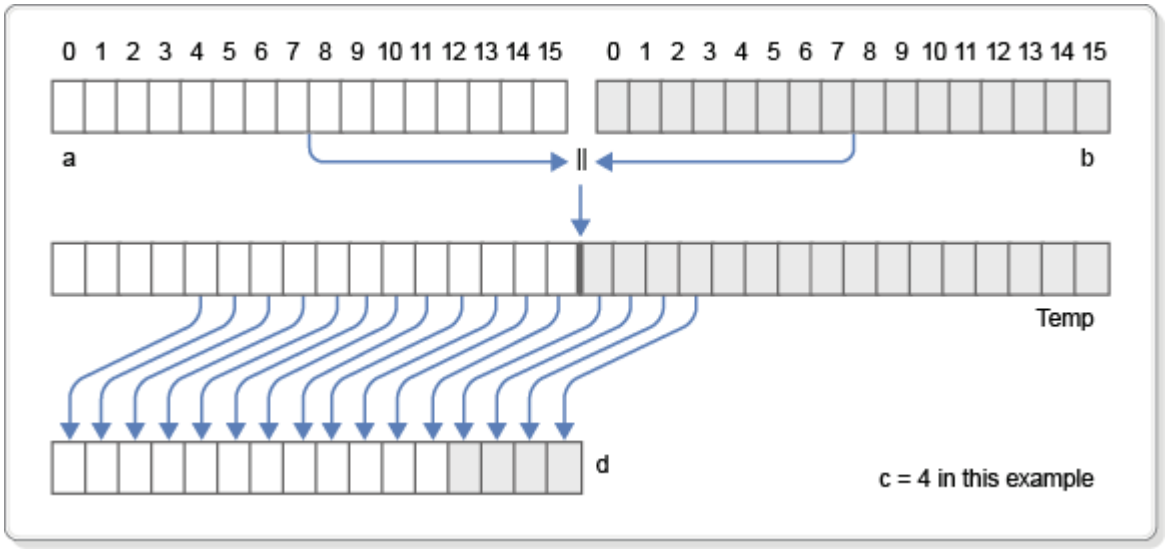


Figure 22. Bit-wise conditional select of vector contents (128-bit)

Table 183. Vector Shift Left Double by Byte				
d	**a**	**b**	**c**	**MIN ARCH**
vector bool char	vector bool char	vector bool char	0 - 15	ARCH(11)
vector unsigned char	vector unsigned char	vector unsigned char		ARCH(11)
vector signed char	vector signed char	vector signed char		ARCH(11)
vector bool short	vector bool short	vector bool short		ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short		ARCH(11)
vector signed short	vector signed short	vector signed short		ARCH(11)
vector bool int	vector bool int	vector bool int		ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int		ARCH(11)
vector signed int	vector signed int	vector signed int		ARCH(11)
vector bool long long	vector bool long long	vector bool long long		ARCH(11)
vector unsigned long long	vector unsigned long long	vector unsigned long long		ARCH(11)
vector signed long long	vector signed long long	vector signed long long		ARCH(11)
vector float	vector float	vector float		ARCH(12)
vector double	vector double	vector double		ARCH(11)

vec_sldw: Vector Shift Left Double by Word

```
d = vec_sldw(a, b, c)
```

Returns a vector by concatenating a and b, and then left shifts the result vector by multiples of 4 bytes. c specifies the offset for the shifting operation. After left-shifting the concatenated a and b by multiples of 4 bytes specified by c, the function takes the four leftmost 4-byte values and forms the result vector.

Table 184. Vector Shift Left Double by Word				
d	**a**	**b**	**c**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	0 - 3	ARCH(11)
vector signed char	vector signed char	vector signed char		ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short		ARCH(11)
vector signed short	vector signed short	vector signed short		ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int		ARCH(11)
vector signed int	vector signed int	vector signed int		ARCH(11)
vector unsigned long long	vector unsigned long long	vector unsigned long long		ARCH(11)
vector signed long long	vector signed long long	vector signed long long		ARCH(11)

vec_sll: Vector Shift Left

```
d = vec_sll(a, b)
```

Performs a left shift for a vector by a given number of bits. Each element of the result is obtained by shifting the corresponding element of a left by the number of bits specified by the last 3 bits of every byte of b. The bits that are shifted out are replaced by zeros.

Note: The low-order 3 bits of all byte elements in b must be the same, otherwise the result is undefined.

Table 185. Vector Shift Left

d	a	b	MIN ARCH
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11) 1
vector signed char	vector signed char		ARCH(11) 1
vector unsigned short	vector unsigned short		ARCH(11) 1
vector signed short	vector signed short		ARCH(11) 1
vector unsigned int	vector unsigned int		ARCH(11) 1
vector signed int	vector signed int		ARCH(11) 1
vector unsigned long long	vector unsigned long long		ARCH(11) 1
vector signed long long	vector signed long long		ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_srab: Vector Shift Right Arithmetic by Byte

```
d = vec_srab(a, b)
```

Performs an algebraic right shift for a vector by a given number of bytes. Each element of the result is obtained by shifting the corresponding element of a right by the number of bytes specified by bits 1-4 of byte element seven of b. The bits that are shifted out are replaced by copies of the most significant bit of the element of a.

Table 186. Vector Shift Right Arithmetic by Byte

d	a	b	MIN ARCH
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)
		vector signed char	ARCH(11)
vector signed char	vector signed char	vector unsigned char	ARCH(11)
		vector signed char	ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11)
		vector signed short	ARCH(11)
vector signed short	vector signed short	vector unsigned short	ARCH(11)
		vector signed short	ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11)
		vector signed int	ARCH(11)
vector signed int	vector signed int	vector unsigned int	ARCH(11)
		vector signed int	ARCH(11)

Table 186. Vector Shift Right Arithmetic by Byte (continued)			
d	**a**	**b**	**MIN ARCH**
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(11)
		vector signed long long	ARCH(11)
vector signed long long	vector signed long long	vector unsigned long long	ARCH(11)
		vector signed long long	ARCH(11)
vector float	vector float	vector unsigned int	ARCH(12)
		vector signed int	ARCH(12)
vector double	vector double	vector unsigned long long	ARCH(11)
		vector signed long long	ARCH(11)

vec_sral: Vector Shift Right Arithmetic

```
d = vec_sral(a, b)
```

Performs an algebraic right shift for a vector by a given number of bits. Each element of the result is obtained by shifting the corresponding element of a right by the number of bits specified by the last 3 bits of every byte of b. The bits that are shifted out are replaced by copies of the most significant bit of the element of a.

Note: The low-order 3 bits of all byte elements in b must be the same, otherwise the result is undefined.

Table 187. Vector Shift Right Arithmetic			
d	**a**	**b**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)
vector signed char	vector signed char		ARCH(11)
vector unsigned short	vector unsigned short		ARCH(11)
vector signed short	vector signed short		ARCH(11)
vector unsigned int	vector unsigned int		ARCH(11)
vector signed int	vector signed int		ARCH(11)
vector unsigned long long	vector unsigned long long		ARCH(11)
vector signed long long	vector signed long long		ARCH(11)
vector bool long long	vector bool long long		ARCH(11)

vec_srb: Vector Shift Right by Byte

```
d = vec_srb(a, b)
```

Performs a right shift for a vector by a given number of bytes. Each element of the result is obtained by shifting the corresponding element of a right by the number of bytes specified by bits 1-4 of byte element seven of b. The bits that are shifted out are replaced by zeros.

Table 188. Vector Shift Right by Byte			
d	**a**	**b**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11)
		vector signed char	ARCH(11)
vector signed char	vector signed char	vector unsigned char	ARCH(11)
		vector signed char	ARCH(11)
vector unsigned short	vector unsigned short	vector unsigned short	ARCH(11)
		vector signed short	ARCH(11)
vector signed short	vector signed short	vector unsigned short	ARCH(11)
		vector signed short	ARCH(11)
vector unsigned int	vector unsigned int	vector unsigned int	ARCH(11)
		vector signed int	ARCH(11)
vector signed int	vector signed int	vector unsigned int	ARCH(11)
		vector signed int	ARCH(11)
vector unsigned long long	vector unsigned long long	vector unsigned long long	ARCH(11)
		vector signed long long	ARCH(11)
vector signed long long	vector signed long long	vector unsigned long long	ARCH(11)
		vector signed long long	ARCH(11)
vector float	vector float	vector unsigned int	ARCH(12)
		vector signed int	ARCH(12)
vector double	vector double	vector unsigned long long	ARCH(11)
		vector signed long long	ARCH(11)

vec_srl: Vector Shift Right

```
d = vec_srl(a, b)
```

Performs a right shift for a vector by a given number of bits. Each element of the result is obtained by shifting the corresponding element of a right by the number of bits specified by the last 3 bits of every byte of b. The bits that are shifted out are replaced by zeros.

Note: The low-order 3 bits of all byte elements in b must be the same, otherwise the result is undefined.

Table 189. Vector Shift Right			
d	**a**	**b**	**MIN ARCH**
vector unsigned char	vector unsigned char	vector unsigned char	ARCH(11) 1
vector signed char	vector signed char		ARCH(11) 1
vector unsigned short	vector unsigned short		ARCH(11) 1
vector signed short	vector signed short		ARCH(11) 1
vector unsigned int	vector unsigned int		ARCH(11) 1
vector signed int	vector signed int		ARCH(11) 1
vector unsigned long long	vector unsigned long long		ARCH(11) 1
vector signed long long	vector signed long long		ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

Rounding and Conversion

This section describes vector built-in functions for rounding and conversion.

vec_ceil: Vector Ceiling

```
d = vec_ceil(a)
```

Returns a vector containing the smallest representable floating-point integral values greater than or equal to the values of the corresponding elements of the given vector.

Note: `vec_ceil` provides the same functionality as `vec_roundp`, except that `vec_ceil` could trigger the IEEE-inexact exception.

Table 190. Vector Ceiling		
d	**a**	**MIN ARCH**
vector float	vector float	ARCH(12) 2
vector double	vector double	ARCH(11) 2
Note: This prototype has slightly different semantics than that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4. In Enterprise Metal C for z/OS, `vec_ceil` could trigger the IEEE-inexact exception.		

Related reference

"vec_roundp: Vector Round toward Positive Infinity" on page 137

vec_double: Vector Convert from long long to double

```
d = vec_double(a)
```

Converts a vector of long long integers into a vector of double-precision numbers.

Note: Current BFP rounding mode is used on the conversion.

Table 191. Vector Convert from Logical		
d	**a**	**MIN ARCH**
vector double	vector signed long long	ARCH(11) 1
	vector unsigned long long	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.		

vec_doublee: Vector Convert from float (even elements) to double

```
d = vec_doublee(a)
```

Converts an input vector to a vector of double-precision numbers. Elements 0 and 2 from a are converted to double-precision numbers and placed in elements 0 and 1 in d respectively.

Table 192. Vector Load Lengthened		
d	**a**	**MIN ARCH**
vector double	vector float	ARCH(12) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.		

vec_extend_s64: Vector Sign Extend to Doubleword

```
d = vec_extend_s64(a)
```

Returns a vector with sign-extended on the rightmost element-sized sub-element of each doubleword.

Table 193. Extend Sign to Doubleword		
d	**a**	**MIN ARCH**
vector signed long long	vector signed char	ARCH(11)
vector signed long long	vector signed short	ARCH(11)
vector signed long long	vector signed int	ARCH(11)

vec_floate: Vector Convert from double to float (even elements)

```
d = vec_floate(a)
```

Converts an input vector to a vector of single-precision numbers. The even-numbered target elements are obtained by converting the source elements to single-precision numbers.

Table 194. Vector Load Rounded		
d	**a**	**MIN ARCH**
vector float	vector double	ARCH(12) 1

Table 194. Vector Load Rounded (continued)		
d	**a**	**MIN ARCH**
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.		

vec_floor: Vector Floor

```
d = vec_floor(a)
```

Returns a vector containing the largest representable floating-point integral values less than or equal to the values of the corresponding elements of the given vector.

Note: `vec_floor` provides the same functionality as `vec_roundm`, except that `vec_floor` could trigger the IEEE-inexact exception.

Table 195. Vector Floor		
d	**a**	**MIN ARCH**
vector float	vector float	ARCH(12) 2
vector double	vector double	ARCH(11) 2
Note: This prototype has slightly different semantics than that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4. In Enterprise Metal C for z/OS, `vec_floor` could trigger the IEEE-inexact exception.		

Related reference

"vec_roundm: Vector Round toward Negative Infinity" on page 136

vec_rint: Vector Round to Integer

```
d = vec_rint(a)
```

Returns a vector by using the current rounding mode to round every double-precision floating-point element in the given vector to integer.

Note: `vec_rint` provides the same functionality as `vec_roundc`, except that `vec_rint` could trigger the IEEE-inexact exception.

Table 196. Vector Round to Integer		
d	**a**	**MIN ARCH**
vector float	vector float	ARCH(12) 2
vector double	vector double	ARCH(11) 2
Note: This prototype has slightly different semantics than that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4. In Enterprise Metal C for z/OS, `vec_rint` could trigger the IEEE-inexact exception.		

Related reference

"vec_roundc: Vector Round to Current" on page 136

vec_round: Vector Round to Nearest

```
d = vec_round(a)
```

Returns a vector containing the rounded values to the nearest representable floating-point integer, using IEEE round-to-nearest rounding, of the corresponding elements of the given vector.

Note: IEEE-inexact exception is suppressed.

Table 197. Vector Round to Nearest

d	a	MIN ARCH
vector float	vector float	ARCH(12) 1
vector double	vector double	ARCH(11) 1

Note:

1. This prototype has slightly different semantics than that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_roundc: Vector Round to Current

```
d = vec_roundc(a)
```

Returns a vector by using the current rounding mode to round every double-precision floating-point element in the given vector to integer.

Note: `vec_roundc` provides the same functionality as `vec_rint`, except that `vec_roundc` does not trigger the IEEE-inexact exception.

Table 198. Vector Round to Current

d	a	MIN ARCH
vector float	vector float	ARCH(12)
vector double	vector double	ARCH(11)

Related reference

"vec_rint: Vector Round to Integer" on page 135

vec_roundm: Vector Round toward Negative Infinity

```
d = vec_roundm(a)
```

Returns a vector containing the largest representable floating-point integral values less than or equal to the values of the corresponding elements of the given vector.

Note: `vec_roundm` provides the same functionality as `vec_floor`, except that `vec_roundm` would not trigger the IEEE-inexact exception.

Table 199. Vector Round toward Negative Infinity

d	a	MIN ARCH
vector float	vector float	ARCH(12)
vector double	vector double	ARCH(11)

Related reference

"vec_floor: Vector Floor" on page 135

vec_roundp: Vector Round toward Positive Infinity

```
d = vec_roundp(a)
```

Returns a vector containing the smallest representable floating-point integral values greater than or equal to the values of the corresponding elements of the given vector.

Note: `vec_roundp` provides the same functionality as `vec_ceil`, except that `vec_roundp` would not trigger the IEEE-inexact exception.

Table 200. Vector Round toward Positive Infinity

d	a	MIN ARCH
vector float	vector float	ARCH(12)
vector double	vector double	ARCH(11)

Related reference

"vec_ceil: Vector Ceiling" on page 133

vec_roundz: Vector Round toward Zero

```
d = vec_roundz(a)
```

Returns a vector containing the truncated values of the corresponding elements of the given vector. Each element of the result contains the value of the corresponding element of a, truncated to an integral value.

Note: `vec_roundz` provides the same functionality as `vec_trunc`, except that `vec_roundz` would not trigger the IEEE-inexact exception.

Table 201. Vector Round toward Zero

d	a	MIN ARCH
vector float	vector float	ARCH(12)
vector double	vector double	ARCH(11)

vec_signed: Vector Convert double to signed long long

```
d = vec_signed(a)
```

Converts a vector of double-precision numbers to a vector of signed integers, rounding toward 0. Each element of d is converted from the corresponding element of a.

Table 202. Vector Convert double to signed long long

d	a	MIN ARCH
vector signed long long	vector double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_trunc: Vector Truncate

```
d = vec_trunc(a)
```

Returns a vector containing the truncated values of the corresponding elements of the given vector. Each element of the result contains the value of the corresponding element of a, truncated to an integral value.

Note: `vec_trunc` provides the same functionality as `vec_roundz`, except that `vec_trunc` could trigger the IEEE-inexact exception.

Table 203. Vector Truncate

d	a	MIN ARCH
vector float	vector float	ARCH(12) 2
vector double	vector double	ARCH(11) 2

Note:

2. This prototype has slightly different semantics than that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4. In Enterprise Metal C for z/OS, `vec_trunc` could trigger the IEEE-inexact exception.

Related reference

"vec_roundz: Vector Round toward Zero" on page 137

vec_unsigned: Vector Convert double to unsigned long long

```
d = vec_unsigned(a)
```

Converts a vector of double-precision numbers to a vector of unsigned integers, rounding toward 0. Each element of d is converted from the corresponding element of a.

Table 204. Vector Convert double to unsigned long long

d	a	MIN ARCH
vector unsigned long long	vector double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

Test

This section describes vector built-in functions for testing.

vec_fp_test_data_class: Vector Floating-Point Test Data Class

```
d = vec_fp_test_data_class (a, b, c)
```

Performs a test of the BFP element class on the vector element a, based on the specified condition b, using the VECTOR FP TEST DATA CLASS IMMEDIATE (VFTCIDB) instruction. The condition code set by the VFTCIDB instruction is returned through c.

d represents the first operand in the instruction.

a represents the second operand in the instruction.

b represents the third operand in the instruction. You can use the __VEC_CLASS_FP_* macros that are defined in `builtins.h` as the argument for this operand.

Table 205. Vector Floating-Point Test Data Class

d	a	b	c	MIN ARCH
vector bool int	vector float	0 - 4095	int *	ARCH(12)
vector bool long long	vector double			ARCH(11)

The following macros define the constants that can be used as the argument b of `vec_fp_test_data_class`. These macros are defined in `builtins.h`.

```
#define __VEC_CLASS_FP_ZERO_P (1 << 11)

#define __VEC_CLASS_FP_ZERO_N (1 << 10)

#define __VEC_CLASS_FP_ZERO (__VEC_CLASS_FP_ZERO_P | __VEC_CLASS_FP_ZERO_N)

#define __VEC_CLASS_FP_NORMAL_P (1 << 9)

#define __VEC_CLASS_FP_NORMAL_N (1 << 8)

#define __VEC_CLASS_FP_NORMAL (__VEC_CLASS_FP_NORMAL_P | __VEC_CLASS_FP_NORMAL_N)

#define __VEC_CLASS_FP_SUBNORMAL_P (1 << 7)

#define __VEC_CLASS_FP_SUBNORMAL_N (1 << 6)

#define __VEC_CLASS_FP_SUBNORMAL (__VEC_CLASS_FP_SUBNORMAL_P | __VEC_CLASS_FP_SUBNORMAL_N)

#define __VEC_CLASS_FP_INFINITY_P (1 << 5)

#define __VEC_CLASS_FP_INFINITY_N (1 << 4)

#define __VEC_CLASS_FP_INFINITY (__VEC_CLASS_FP_INFINITY_P | __VEC_CLASS_FP_INFINITY_N)

#define __VEC_CLASS_FP_QNAN_P (1 << 3)

#define __VEC_CLASS_FP_QNAN_N (1 << 2)

#define __VEC_CLASS_FP_QNAN (__VEC_CLASS_FP_QNAN_P | __VEC_CLASS_FP_QNAN_N)

#define __VEC_CLASS_FP_SNAN_P (1 << 1)

#define __VEC_CLASS_FP_SNAN_N (1 << 0)

#define __VEC_CLASS_FP_SNAN (__VEC_CLASS_FP_SNAN_P | __VEC_CLASS_FP_SNAN_N)

#define __VEC_CLASS_FP_NAN (__VEC_CLASS_FP_QNAN | __VEC_CLASS_FP_SNAN)

#define __VEC_CLASS_FP_NOT_NORMAL (__VEC_CLASS_FP_NAN | __VEC_CLASS_FP_SUBNORMAL |
__VEC_CLASS_FP_ZERO | __VEC_CLASS_FP_INFINITY)
```

vec_test_mask: Vector Test under Mask

```
d = vec_test_mask(a, b)
```

Returns the condition code set by the Vector Test Under Mask (VTM) instruction. a is the first operand, and b is the second operand on the instruction.

Table 206. Vector Test under Mask			
d	**a**	**b**	**MIN ARCH**
int	vector signed char	vector unsigned char	ARCH(11)
	vector unsigned char		ARCH(11)
	vector signed short	vector unsigned short	ARCH(11)
	vector unsigned short		ARCH(11)
	vector signed int	vector unsigned int	ARCH(11)
	vector unsigned int		ARCH(11)
	vector signed long long	vector unsigned long long	ARCH(11)
	vector unsigned long long		ARCH(11)
	vector float	vector unsigned int	ARCH(12)
	vector double	vector unsigned long long	ARCH(11)

All Predicates

This section describes vector built-in functions for searching and comparing all elements.

vec_all_eq: All Elements Equal

```
d = vec_all_eq(a, b)
```

Tests whether all sets of corresponding elements of the given vectors are equal. The result is 1 if each element of a is equal to the corresponding element of b. Otherwise, the result is 0.

Table 207. All Elements Equal			
d	**a**	**b**	**MIN ARCH**
int	vector bool char	vector bool char	ARCH(11) 1
	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
	vector bool short	vector bool short	ARCH(11) 1
	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
	vector bool int	vector bool int	ARCH(11) 1
	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector bool long long	vector bool long long	ARCH(11) 1
	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1

Table 207. All Elements Equal (continued)			
d	**a**	**b**	**MIN ARCH**
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_all_ge: All Elements Greater Than or Equal

```
d = vec_all_ge(a, b)
```

Tests whether all elements of the first argument are greater than or equal to the corresponding elements of the second argument. The result is 1 if all elements of a are greater than or equal to the corresponding elements of b. Otherwise, the result is 0.

Table 208. All Elements Greater Than or Equal			
d	**a**	**b**	**MIN ARCH**
int	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_all_gt: All Elements Greater Than

```
d = vec_all_gt(a, b)
```

Tests whether all elements of the first argument are greater than the corresponding elements of the second argument. The result is 1 if all elements of a are greater than the corresponding elements of b. Otherwise, the result is 0.

Table 209. All Elements Greater Than			
d	**a**	**b**	**MIN ARCH**
int	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_all_le: All Elements Less Than or Equal

```
d = vec_all_le(a, b)
```

Tests whether all elements of the first argument are less than or equal to the corresponding elements of the second argument. The result is 1 if all elements of a are less than or equal to the corresponding elements of b. Otherwise, the result is 0.

Table 210. All Elements Less Than or Equal			
d	**a**	**b**	**MIN ARCH**
int	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_all_lt: All Elements Less Than

```
d = vec_all_lt(a, b)
```

Tests whether all elements of the first argument are less than the corresponding elements of the second argument. The result is 1 if all elements of a are less than the corresponding elements of b. Otherwise, the result is 0.

Table 211. All Elements Less Than

d	a	b	MIN ARCH
int	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_all_nan: All Elements Not a Number

```
d = vec_all_nan(a)
```

Tests whether each element of the given vector is a NaN. The result is 1 if each element of a is a NaN. Otherwise, the result is 0.

Table 212. All Elements Not a Number

d	a	MIN ARCH
int	vector float	ARCH(12) 1
	vector double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_all_ne: All Elements Not Equal

```
d = vec_all_ne(a, b)
```

Tests whether all sets of corresponding elements of the given vectors are not equal. The result is 1 if each element of a is not equal to the corresponding element of b. Otherwise, the result is 0.

Table 213. All Elements Not Equal			
d	**a**	**b**	**MIN ARCH**
int	vector bool char	vector bool char	ARCH(11) 1
	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
	vector bool short	vector bool short	ARCH(11) 1
	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
	vector bool int	vector bool int	ARCH(11) 1
	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector bool long long	vector bool long long	ARCH(11) 1
	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_all_nge: All Elements Not Greater Than or Equal

```
d = vec_all_nge(a, b)
```

Tests whether each element of the first argument is not greater than or equal to the corresponding element of the second argument. The result is 1 if each element of a is not greater than or equal to the corresponding element of b. Otherwise, the result is 0.

Table 214. All Elements Not Greater Than or Equal			
d	**a**	**b**	**MIN ARCH**
int	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_all_ngt: All Elements Not Greater Than

```
d = vec_all_ngt(a, b)
```

Tests whether each element of the first argument is not greater than the corresponding element of the second argument. The result is 1 if each element of a is not greater than the corresponding element of b. Otherwise, the result is 0.

Table 215. All Elements Not Greater Than			
d	**a**	**b**	**MIN ARCH**
int	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_all_nle: All Elements Not Less Than or Equal

```
d = vec_all_nle(a, b)
```

Tests whether each element of the first argument is not less than or equal to the corresponding element of the second argument. The result is 1 if each element of a is not less than or equal to the corresponding element of b. Otherwise, the result is 0.

Table 216. All Elements Not Less Than or Equal			
d	**a**	**b**	**MIN ARCH**
int	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_all_nlt: All Elements Not Less Than

```
d = vec_all_nlt(a, b)
```

Tests whether each element of the first argument is not less than the corresponding element of the second argument. The result is 1 if each element of a is not less than the corresponding element of b. Otherwise, the result is 0.

Table 217. All Elements Not Less Than			
d	**a**	**b**	**MIN ARCH**
int	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_all_numeric: All Elements Numeric

```
d = vec_all_numeric(a)
```

Tests whether each element of the given vector is numeric (not a NaN). The result is 1 if each element of a is numeric (not a NaN). Otherwise, the result is 0.

Table 218. All Elements Numeric		
d	**a**	**MIN ARCH**
int	vector float	ARCH(12) 1
	vector double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

Any Predicates

This section describes vector built-in functions for searching and comparing any elements.

vec_any_eq: Any Element Equal

```
d = vec_any_eq(a, b)
```

Tests whether any set of corresponding elements of the given vectors are equal. The result is 1 if any element of a is equal to the corresponding element of b. Otherwise, the result is 0.

Table 219. Any Element Equal			
d	**a**	**b**	**MIN ARCH**
int	vector bool char	vector bool char	ARCH(11) 1
	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
	vector bool short	vector bool short	ARCH(11) 1
	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
	vector bool int	vector bool int	ARCH(11) 1
	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector bool long long	vector bool long long	ARCH(11) 1
	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_any_ge: Any Element Greater Than or Equal

```
d = vec_any_ge(a, b)
```

Tests whether any element of the first argument is greater than or equal to the corresponding element of the second argument. The result is 1 if any element of a is greater than or equal to the corresponding element of b. Otherwise, the result is 0.

Table 220. Any Element Greater Than or Equal

d	a	b	MIN ARCH
int	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1

Note:

1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.

vec_any_gt: Any Element Greater Than

```
d = vec_any_gt(a, b)
```

Tests whether any element of the first argument is greater than the corresponding element of the second argument. The result is 1 if any element of a is greater than the corresponding element of b. Otherwise, the result is 0.

Table 221. Any Element Greater Than

d	a	b	MIN ARCH
int	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1

Table 221. Any Element Greater Than (continued)			
d	**a**	**b**	**MIN ARCH**
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_any_le: Any Element Less Than or Equal

```
d = vec_any_le(a, b)
```

Tests whether any element of the first argument is less than or equal to the corresponding element of the second argument. The result is 1 if any element of a is less than or equal to the corresponding element of b. Otherwise, the result is 0.

Table 222. Any Element Less Than or Equal			
d	**a**	**b**	**MIN ARCH**
int	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_any_lt: Any Element Less Than

```
d = vec_any_lt(a, b)
```

Tests whether any element of the first argument is less than the corresponding element of the second argument. The result is 1 if any element of a is less than the corresponding element of b. Otherwise, the result is 0.

Table 223. Any Element Less Than			
d	**a**	**b**	**MIN ARCH**
int	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_any_ne: Any Element Not Equal

```
d = vec_any_ne(a, b)
```

Tests whether any set of corresponding elements of the given vectors are not equal. The result is 1 if any element of a is not equal to the corresponding element of b. Otherwise, the result is 0.

Table 224. Any Element Not Equal			
d	**a**	**b**	**MIN ARCH**
int	vector bool char	vector bool char	ARCH(11) 1
	vector signed char	vector signed char	ARCH(11) 1
	vector unsigned char	vector unsigned char	ARCH(11) 1
	vector bool short	vector bool short	ARCH(11) 1
	vector signed short	vector signed short	ARCH(11) 1
	vector unsigned short	vector unsigned short	ARCH(11) 1
	vector bool int	vector bool int	ARCH(11) 1
	vector signed int	vector signed int	ARCH(11) 1
	vector unsigned int	vector unsigned int	ARCH(11) 1
	vector bool long long	vector bool long long	ARCH(11) 1
	vector signed long long	vector signed long long	ARCH(11) 1
	vector unsigned long long	vector unsigned long long	ARCH(11) 1
	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1

Table 224. Any Element Not Equal (continued)			
d	**a**	**b**	**MIN ARCH**
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_any_nan: Any Element Not a Number

```
d = vec_any_nan(a)
```

Tests whether any element of the given vector is a NaN. The result is 1 if any element of a is a NaN. Otherwise, the result is 0.

Table 225. Any Element Not a Number		
d	**a**	**MIN ARCH**
int	vector float	ARCH(12) 1
	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.		

vec_any_nge: Any Element Not Greater Than or Equal

```
d = vec_any_nge(a, b)
```

Tests whether any element of the first argument is not greater than or equal to the corresponding element of the second argument. The result is 1 if any element of a is not greater than or equal to the corresponding element of b. Otherwise, the result is 0.

Table 226. Any Element Not Greater Than or Equal			
d	**a**	**b**	**MIN ARCH**
int	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_any_ngt: Any Element Not Greater Than

```
d = vec_any_ngt(a, b)
```

Tests whether any element of the first argument is not greater than the corresponding element of the second argument. The result is 1 if any element of a is not greater than the corresponding element of b. Otherwise, the result is 0.

Table 227. Any Element Not Greater Than			
d	**a**	**b**	**MIN ARCH**
int	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_any_nle: Any Element Not Less Than or Equal

```
d = vec_any_nle(a, b)
```

Tests whether any element of the first argument is not less than or equal to the corresponding element of the second argument. The result is 1 if any element of a is not less than or equal to the corresponding element of b. Otherwise, the result is 0.

Table 228. Any Element Not Less Than or Equal			
d	**a**	**b**	**MIN ARCH**
int	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_any_nlt: Any Element Not Less Than

```
d = vec_any_nlt(a, b)
```

Tests whether any element of the first argument is not less than the corresponding element of the second argument. The result is 1 if any element of a is not less than the corresponding element of b. Otherwise, the result is 0.

Table 229. Any Element Not Less Than			
d	**a**	**b**	**MIN ARCH**
int	vector float	vector float	ARCH(12) 1
	vector double	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.			

vec_any_numeric: Any Element Numeric

```
d = vec_any_numeric(a)
```

Tests whether any element of the given vector is numeric (not a NaN). The result is 1 if any element of a is numeric (not a NaN). Otherwise, the result is 0.

Table 230. Any Element Numeric		
d	**a**	**MIN ARCH**
int	vector float	ARCH(12) 1
	vector double	ARCH(11) 1
Note: 1. This prototype has the exact same semantics as that in the OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI, Revision 1.4.		

Defining vector built-in functions from operators

The following function-like macros can be used to define some vector built-in functions, available on the XL C/C++ compilers for some other platforms, from the operators:

```
#define vec_neg(a) (-(a)) // Vector Negate
#define vec_add(a, b) ((a) + (b)) // Vector Add
#define vec_sub(a, b) ((a) - (b)) // Vector Subtract
#define vec_mul(a, b) ((a) * (b)) // Vector Multiply
#define vec_div(a, b) ((a) / (b)) // Vector Divide
#define vec_and(a, b) ((a) & (b)) // Vector AND
#define vec_or(a, b) ((a) | (b)) // Vector OR
#define vec_xor(a, b) ((a) ^ (b)) // Vector XOR
#define vec_sl(a, b) ((a) << (b)) // Vector Shift Left
#define vec_sra(a, b) ((a) >> (b)) // Vector Shift Right
#define vec_sr(a, b) ((a) >> (b)) // Vector Shift Right Algebraic
#define vec_slo(a, b) vec_slb(a, (b) << 64) // Vector Shift Left by Octet
#define vec_sro(a, b) vec_srb(a, (b) << 64) // Vector Shift Right by Octet
```

Note: The `vec_sra` macro definition must only be used with first parameter having a `vector signed` types. Similarly, the `vec_sr` macro definition must only be used for `vector unsigned` types, to have the correct bits inserted on the shifted out bits.

Part 2. Performance optimization

This part describes guidelines for improving the performance of your Enterprise Metal C for z/OS application. Performance improvement can be achieved through coding and compiling. The following chapters discuss guidelines for these three areas:

Chapter 4. Improving program performance

This information discusses coding guidelines that improve the performance of a C application. While they are most effective when creating new code, these guidelines can also provide a gradual performance improvement when they are consistently used when porting or fixing areas of the code. The guidelines cover the following topics:

Writing code for performance

When you write code, it is a good practice to write it so that you can understand it when you simply read it on a printed page or on a screen, without having to refer to anything else. If the code is simple and concise, both the programmer and the compiler can understand it easily. Code that is easy for the compiler to understand is also easy for it to optimize. If you follow this practice you might not only create code that performs well on execution, you might also create code that compiles more quickly.

If you follow the guidelines in this information, you will create code that performs well on execution and can be compiled efficiently.

ANSI aliasing rules

You must indicate whether your source code conforms to the ANSI aliasing rules when you use the IPA or the OPT(2) (or above) compiler options. If the code does not conform to the rules, it must be compiled with NOANSIALIAS. Incorrect use of these options might generate bad code.

Note: The compiler expects that the source code conforms to the ANSI aliasing rules when the ANSIALIAS option is used. This option is on by default.

The ANSI aliasing rules are part of the ISO C Standard, and state that a pointer can be dereferenced only to an object of the same type or compatible type. Because the Enterprise Metal C for z/OS compiler follows these rules during optimization, the developer must create code that conforms to the rules.

Note: The common coding practice of casting a pointer to an incompatible type and then dereferencing it violates ANSI aliasing rules.

When you are using ANSI aliasing, you can cast an `int` pointer only to the types described in Table 231 on page 156 then dereference it to access the object it points to. Corresponding casts apply to other types.

Table 231. Examples of acceptable alias types

Type	Reason for acceptance
`int`	This is the declared type of the object.
`const int`, or `volatile int`, or `restrict int`, or any combination of these qualifiers	These types are the qualified version of the declared type of the object.
`signed int` or `unsigned int`	This is a signed or unsigned type corresponding to the declared type of the object.
`const unsigned int` or `volatile unsigned int`	These types are the signed or unsigned types corresponding to a qualified version of the declared type of the object.
`struct myfunc { unsigned int bar; };`	This is an aggregate or union type that includes one of the aforementioned types among its members. This can include, recursively, a member of a subaggregator-contained union.
`char`, `unsigned char`, or `signed char`	The `char` pointers are an exception to the rules, as a `char` pointer can be used to point to and dereferenced to access a variable of any type. For example, the address passed to `memcpy` may be any pointer type.

Conversely, your code breaks the aliasing rules if it casts a `float` to an `int` and then assigns it to the `int` pointer and dereferences that.

In C11, the typeless memory returned by `malloc` etc. receives the "effective type" of the first access to it. For example, if the address returned by `malloc` is cast to an `int*` pointer and that is dereferenced to store an initial value, then that memory's effective type becomes `int`, and only pointers compatible with `int` can be used to access it.

For more information, see ANSIALIAS | NOANSIALIAS in .

You can cast and mix data types as long as you are careful how you intermix values and their pointers in your code. The compiler follows the ANSI aliasing rules to determine:

- Which variables must be stored into memory before you read a value through a pointer
- Which variables must be updated from memory after you have updated a value through a pointer

When you use the NOANSIALIAS option, the compiler generates code to accommodate worst-case assumptions (for example, that any variable could have been updated by the store through a pointer). This means that every variable (local and global) must be stored in memory to ensure that any value can be read through a pointer. This severely limits the potential for optimization.

```
int ei1;
float ef1;
int *eip1;
float *efp1;

float exmp1 ()
{
   ef1 = 3.0;
   ei1=5;
   *efp1 = ef1;
   *eip1 = ei1;
   return *efp1;
}
```

Table 232 on page 157 shows the difference between code generated with, and without, ANSI aliasing.

Table 232. Comparison of code generated with the ANSIALIAS and NOANSIALIAS options

ANSIALIAS RENT and OPT(2)

```
* {
*  ef1 = 3.0;
     L      r4,=A(@CONSTANT_AREA)(,r3,94)
     L      r2,=Q(EF1)(,r3,98)
     LD     f0,+CONSTANT_AREA(,r4,0)
     L      r14,_CEECAA_(,r12,500)
     L      r15,=Q(EFP1)(,r3,102)
     L      r4,=Q(EIP1)(,r3,106)
     L      r1,#retvalptr_1(,r1,0)
     STE    f0,ef1(r2,r14,0)
     L      r15,efp1(r15,r14,0)

*  ei1=5;
     L      r2,=Q(EI1)(,r3,110)
     LA     r0,
     L      r4,eip1(r4,r14,0)

*  *efp1 = ef1;
     STE    f0,(*)float(,r15,0)
     ST     r0,ei1(r2,r14,0)

*  *eip1 = ei1;
     ST     r0,(*)int(,r4,0)

*  return *efp1;
     STD    f0,#retval_1(,r1,0)
*  }
```

NOANSIALIAS RENT and OPT(2)

```
* {
*  ef1 = 3.0;
     L      r2,=A(@CONSTANT_AREA)(,r3,110)
     L      r14,_CEECAA_(,r12,500)
     L      r4,=Q(EF1)(,r3,114)
     L      r15,=Q(EFP1)(,r3,118)
     LD     f0,+CONSTANT_AREA(,r2,0)

*  ei1=5;
     L      r2,=Q(EI1)(,r3,122)
     STE    f0,ef1(r4,r14,0)

*  *efp1 = ef1;
     L      r4,efp1(r15,r14,0)

*  *eip1 = ei1;
     L      r5,=Q(EIP1)(,r3,126)
     LA     r0,5
     ST     r0,ei1(r2,r14,0)
     STE    f0,(*)float(,r4,0)
     L      r4,eip1(r5,r14,0)
     L      r0,ei1(r2,r14,0)

*  return *efp1;
     L      r1,#retvalptr_1(,r1,0)
     ST     r0,(*)int(,r4,0)
     L      r14,efp1(r15,r14,0)
     SDR    f0,f0
     LE     f0,(*)float(,r14,0)
     STD    f0,#retval_1(,r1,0)
*  }
```

- In the ANSIALIAS case:
  - `f0`, loaded with 3.0, is used whenever referring to `ef1` or `efp1`
  - `r0` is loaded with the value of 5, which is used for `ei` and `eip`
- In the NOANSIALIAS case, the loads and stores are always done. This removes opportunities for optimizations. For example, if `a + b + c` were used instead of 3.0 and `ef1`, saving through the pointer might have updated a, b, or c, and therefore you cannot common at all, and many more reloads.
- ANSIALIAS would not help if all the floats were also integers
- There is a group of problems that occurs when the ANSIALIAS option is used to compile code that does not conform to ANSI-aliasing rules (for example, when it casts a variable to a non-ANSI-aliasing type and then assigns the address of the value to a pointer for later use). If the ANSIALIAS option is in effect (it is the default) when a value is used through a pointer, the compiler might not reload the pointer value when the original value is updated, and the value might be stale when it is read.

Using ANSI aliasing rules

Your programs are likely to perform better if you follow these guidelines:

- Use ANSI aliasing whenever possible.
- Declare constant variables with `const`.

```
ggPoint3 operator*(const ggHAffineMatrix3 &m
, const ggPoint3 &p)
 {
  return ggPoint3(
    m.e[0][0] * p.x() + m.e[0][1] * p.y() + m.e[0][2] * p.z() + m.e[0][3],
    m.e[1][0] * p.x() + m.e[1][1] * p.y() + m.e[1][2] * p.z() + m.e[1][3],
    m.e[2][0] * p.x() + m.e[2][1] * p.y() + m.e[2][2] * p.z() + m.e[2][3]
    );
}
```

- Whenever their values cannot change, qualify pointers and their targets as constants, ensuring that you mark the appropriate part as `const`.
  - If only the pointer is constant, you can use a statement that is similar to the following:

    ```
    int * const i = p /* a constant pointer to an integer that may vary */
    ```

  - If only the target is constant, use a statement similar to either of the following:

    ```
    int const * i = p /* a variable pointer to a constant integer */
    const int * i = p /* a variable pointer to a constant integer */
    ```

  - If both the target integer and the pointer are constants, use a statement similar to either of the following:

    ```
    const int * const i = &p;  /* a constant pointer to a constant integer  */
    int const * const i = &p;  /* a constant pointer to a constant integer  */
    ```

- Use the ROCONST compiler option. This option causes the compiler to treat variables that are defined as `const` as if they are read-only. In some cases, these variables will be stored in read-only memory. For more information, see “ROCONST” on page 181.
- *For global variables initialized to large read-only arrays or strings:* Use a #pragma variable to ensure that they are implemented as read-only csects. This prevents them from being initialized at load time.

  Example: For large initialized arrays

  ```
  # pragma variable (arrayname, norent)
  ```

- *In a read-only situation:* If you are using the value through a pointer, use a temporary automatic variable. The difference in the source code is significant, as shown in the following table:

Table 233. Example of using temporaries to remove aliasing effects

ANSIALIAS RENT and OPT(2)	**NOANSIALIAS RENT and OPT(2)**
`` ... while (hot_loop < hot_loop_end) { hot_loop = hot_loop + myfunc- >increment; fun[x] = hot_loop*myfunc->expansion; } } ```	``` { ... increment = myfunc->increment; expansion = myfunc->expansion; while (hot_loop < hot_loop_end) { hot_loop = hot_loop + increment; fun[x] = hot_loop*expansion; } ```

Using variables

When choosing variables and data structures for your application, keep the following guidelines in mind:

- Use local variables, preferably automatic variables, as often as possible.

  The compiler can accurately analyze the use of local variables, while it has to make several worst-case assumptions about global variables, which hinders optimizations. For example, if you code a function

that uses external variables, and calls several external functions, the compiler assumes that every call to an external function could change the value of every external variable.

- If none of the function calls affect the global variables being used and you have to read them frequently with function calls interspersed, copy the global variables to local variables and use these local variables to help the compiler perform optimizations that otherwise would not be done.

  Using IPA can improve the performance of code written using global variables, because it coalesces global variables. IPA puts global variables into one or more structures and accesses them using offsets from the beginning of the structures. For more information, see “Using the IPA option” on page 175.

- If you need to share variables only between functions within the same compilation unit, use static variables instead of external variables. Because static variables are visible only in the current source file, they might not have to be reloaded if a call is made to a function in another source file.

  Organize your source code so references to a given set of externally defined variables occur only in one source file, and then use static variables instead of external variables.

  In a file with several related functions and static variables, the compiler can group the variables and functions together to improve locality of reference.

  Use a local static variable instead of an external variable or a variable defined outside the scope of a function.

  The **#pragma isolated_call** preprocessor directive can improve the runtime performance of optimized code by allowing the compiler to make fewer assumptions about the references to external and static variables. For more information, see #pragma isolated_call in .

  Coalescing global variables causes variables that are frequently used together to be mapped close together in memory. This strategy improves performance in the same way that changing external variables to static variables does.

- Group external data into structures (all elements of an external structure use the same base address) or arrays wherever it makes sense to do so.

  Before it can access an external variable, the compiler has to make an extra memory access to obtain the variable’s address. The compiler removes extraneous address loads, but this means that the compiler has to use a register to keep the address.

  Using many external variables simultaneously requires many registers, thereby causing spilling of registers to storage. If you group variables into structures then it can use a single variable to keep the base address of the structure and use offsets to access individual items. This reduces register pressure and improves overall performance, especially in programs compiled with the RENT option.

  The compiler treats register variables the same way it treats automatic variables that do not have their addresses taken.

- Minimize the use of pointers.

  Use of pointers inhibits most memory optimizations such as dead store elimination in C.

  You can improve the runtime performance of optimized code by using the **#pragma disjoint** directive to list identifiers that do not share the same physical storage. A similar mechanism that can be used to improve runtime performance of optimized code includes using the C99 restrict qualifier for pointers feature. The restrict type qualifier indicates for the lifetime of the pointer, and only it or a value directly derived from it will be used to access the object to which it points. For more information, see #pragma disjoint and The restrict type qualifier in .

Passing function arguments

When writing code for optimization, it is usually better to pass a value as an argument to a function than to let the function take the value from a global variable. Global variables might have to be stored before a value is read from a pointer or before a function call is made. Global variables might have to be reloaded after function calls, or stored through a pointer. For more information, see “Using ANSI aliasing rules” on page 157 and “Using variables” on page 158.

The **#pragma isolated_call** preprocessor directive lists functions that do not modify global storage. You can use it to improve the runtime performance of optimized code. For more information, see #pragma isolated_call in .

Linkage convention or how arguments are passed is not specified in the C language, but is defined by the platform. Compilers in general follow the calling convention as described by the Application Binary Interface (ABI). An ABI can define more than one linkage due to performance considerations; for example, the XPLINK on the z/OS platform. To correctly invoke a function, the arguments passed must match the parameters as defined in the function definition. For example, if you pass a pointer argument to a function expecting an integer, the code generated by the compiler for the call and for the function definition may not match (see the note at the end of this topic).

As the following example shows, you can declare a function without providing information about the number and types of its parameters.

```
int func();
...
int a;
func(a);
...
int func(p)
  void *p;
{
  ...
}
```

Because the function declaration has no parameter information, the compiler is not required to diagnose parameter mismatch. You can call this function, passing it any number of arguments of any type, but the compilation will not be guaranteed to work if the function is not defined to receive the arguments as passed, due to differences in linkage conventions. In the worse case, when the compiler attempts inlining of such ill-formed function calls, it may get into an unrecoverable condition and the compilation is halted.

Note: Such a mismatch may sometimes turn out not to be an issue, depending on the ABI; for example, if the ABI happens to allow both pointers and integers passed using general purpose registers. Even in this case, there is no guarantee that the optimized code would work as expected due to ambiguous information received by the compiler.

Coding expressions

When coding expressions, consider the following recommendations:

- When components of an expression are duplicate expressions, code them either at the left end of the expression or within parentheses, as shown in the following example.

```
a = b*(x*y*z);            /* Duplicates recognized */
c = x*y*z*d;
e = f + (x + y);
g = x + y + h;

a = b*x*y*z;              /* No duplicates recognized */
c = x*y*z*d;
e = f + x + y;
g = x + y + h;
```

  The compiler can recognize x*y*z and x + y as duplicate expressions when they are coded in parentheses or coded at the left end of the expression.

  It is the best practice to avoid using pointers as much as possible within high-usage or other performance-critical code.

  Note: The compiler might not be able to optimize duplicate expressions if either of the following are true:

  – The address of any of the variables is already taken

  – Pointers are involved in the computation

- When components of an expression in a loop are constant, code the constant expressions either at the left end of the expression or within parentheses.

  The following example shows the difference in evaluation when c, d, and e are constant and v, w, and x are variable.

```
v*w*x*(c*d*e);       /* Constant expressions recognized */
c + d + e + v + w + x;

v*w*x*c*d*e;         /* Constant expressions not recognized */
v + w + x + c + d + e;
```

Coding conversions

Avoid forcing the compiler to convert numbers between integer and floating-point internal representations. Conversions require several instructions, including some double-precision floating-point arithmetic. When you must use mixed-mode arithmetic, code the integral, floating-point, and decimal arithmetic in separate computations wherever possible. Figure 23 on page 161 shows an example.

```
/* this example shows how numeric conversions are done */

int main(void)
{
   int i;
   float array[10]={1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0}
   float x = 1.0;
   for (i = 0; i < 10; i++)
      {
       array[i] = array[i]*x;  /* No conversions needed */
       x = x + 1.0;
      }

   for (i = 1; i <= 9; i++)
      array[i] = array[i]*i;  /* Conversions may be needed */

   return(0);
}
```

Figure 23. Numeric conversions example

Arithmetical considerations

Wherever possible, use multiplication rather than division. For example,

```
x*(1.0/3.0);  /*  1.0/3.0 is evaluated at compile time  */
```

produces faster code than:

```
x/3.0;
```

If you divide many values by the same number in your code: Assign the divisor's reciprocal to a temporary variable and then multiply by that variable.

Using loops and control constructs

For the `for`-loop index variable:

- Use a `long` type variable whenever possible. Under ILP32, `long` and `int` are equivalent, but `long` is better for portability to an LP64 environment.
- Use the `auto` or `register` storage class over the `extern` or `static` storage class.

- If you use an `enum` variable, expand the variable to be a fullword by using the ENUMSIZE compiler option or by placing a large defined value at the end of your `enum` variable, as follows:

```
enum animals {
    ant
    cat,
    dog,
    robin,
    last_animal = INT_MAX;
};
```

- Do not use the address operator (&) on the index.
- The index should not be a member of a `union`.

For `if` statements:

- Order the `if` conditions efficiently; put the most decisive tests first and the most expensive tests last.

  By performing the most common tests first, you increase the efficiency of your code; fewer tests are required to meet the test conditions.

```
if (command.is_classg && command.len == 6  &&
   !strcmp (command.str, "LOGON"))    /* call to strcmp() most expensive */
  logon ();
```

Choosing a data type

Use the `int` data type instead of `char` when performing arithmetic operations.

```
char_var += '0';
int_var  += '0';         /*  better  */
```

A `char` type variable is efficient when you are:

- Assigning a literal to a `char` variable
- Comparing the variable with a `char` literal

For example:

```
char_var = 27;
if (char_var == 'D')
```

Table 234 on page 162 lists analogous data types and shows which data types are more expensive to reference.

Table 234. Referencing data types

More Expensive	Less Expensive
unsigned short	signed short (Although unsigned short is less expensive on many systems, the z/OS implementation of signed short is less expensive.)
signed char	unsigned char
long double	double
Longer decimal	Shorter decimal

For storage efficiency, the compiler packs enumeration variables in 1, 2 or 4 bytes, depending on the largest value of a constant. When performance is critical, expand the size to a fullword either by adding an enumeration constant with a large value or by specifying the ENUMSIZE compiler option. For example:

```
enum byte { land, sea, air, space };

enum word { low, medium, high, expand_to_fullword = INT_MAX };
```

Example that is equivalent to using the ENUMSIZE(INT) compiler option:

```
enum word { low, medium, high };
```

Fullword enumeration variables are preferred as function parameters.

For efficient use of `extern` variables:

- Place scalars ahead of arrays in `extern struct`.
- Copy heavily referenced scalars to `auto` or `register` variables (especially in a loop).

When using `float`:

- When passing variables of type float to a function, an implicit widening to double occurs (which takes time).
- On some machines divisions of type float are faster than those of type double.

When using bit fields, be aware that:

- Even though the compiler supports a bit field spanning more than 4 bytes, the cost of referencing it is higher.
- An `unsigned` bit field is preferred over a `signed` bit field.
- A bit field used to store integer values should have a length of 8, 16, or 24 bits and be on a byte boundary.

```
    struct {    unsigned   xval  :8,
                           xbool :1,
                           xmany :6,
                           xset  :1;
    } b;

    if (b.xval == 3)
⋮
    if (b.xmany + 5 == x)    /* inefficient because it does not */
                             /* fall on a byte boundary         */
⋮
    if (b.xbool)
⋮
```

Using #pragmas

Table 235 on page 163 describes #pragmas that can affect performance. For information about using each pragma, see .

Table 235. Pragmas that affect performance

Name	Description
#pragma disjoint	Lists identifiers that do not share the same physical storage, which provides more opportunities for optimizations.
#pragma execution_frequency	Marks program source code that you expect will be either very frequently or very infrequently executed.
#pragma export	Selectively exports functions or variables from a DLL module.
#pragma inline	Together with the INLINE compiler option, ensures that frequently used functions are inlined.
#pragma isolated_call	Lists functions that have no side effects (that do not modify global storage). This directive can improve the runtime performance of variables and storage by allowing the compiler to make fewer assumptions about whether external and static variables could be updated.

Table 235. Pragmas that affect performance (continued)

Name	Description
#pragma leaves	Specifies that a function never returns to the instruction following a call to that function. This directive provides information to the compiler that enables it to explore additional opportunities for optimization.
#pragma noinline	This directive can improve pipeline usage and allow more of the used routines to be inlined.
#pragma option_override	Allows you to specify optimization options on a per-routine basis rather than on only a per-compilation basis. It enables you to specify which functions you do not want to optimize while compiling the rest of the program optimized. This directive helps you to isolate which function is causing problems under optimization. The **option_override** pragma can be also used to change the spill size for a function. If the compiler requests that you to increase the spill size for a specific function, you should use the **option_override** pragma, which increases the spill size for all functions in the compile unit and can have a negative performance impact on the generated code. **Note:** The spill size should not be increased unless requested by a compiler message.
#pragma reachable	Declares that the point in the program after the specified function can be the target of a branch from some unknown location. That is, you can reach the instruction after the specified function from a point in your program other than the return statement in the named function. This directive provides information to the compiler that enables it to explore additional opportunities for optimization.
#pragma strings	Indicates if strings should be placed in read-only memory or read/write memory. You can reduce the memory requirements for DLLs by specifying **#pragma strings(readonly)**, so that string literals are not placed in the writable static area. Alternatively, you can also use the ROSTRING compiler option (the default), which informs the compiler that string literals are read-only.
#pragma unroll	Informs the compiler how to perform loop unrolling on the loop body that immediately follows it. The directive works in conjunction with the UNROLL compiler option to provide you with some control over the application of this optimization techique. The pragma directive overrides the “UNROLL” on page 181 or NOUNROLL compiler option in effect for the designated loop.
#pragma variable	Indicates if a named external object is used in reentrant or non-reentrant fashion. If an object is qualified as RENT, its references or its definition will be in the writable static area, which is in modifiable storage. If an object is qualified as NORENT, its references or its definition will be in the code area. You can reduce the memory requirements for DLLs by specifying **#pragma variable(var_name,NORENT)**, so that constant variables are not placed in the writable static area. Alternatively, you can use the ROCONST compiler option to inform the compiler that constant variables are not to be placed in the writable static area.

Chapter 5. Using built-in functions to improve performance

A built-in function is inline code that is generated in place of an actual function call. The compiler will generate inline code for built-in functions, if the appropriate header files are included in the source code.

If you have included the header files but you want to call either the library version of the function or your own version, enclose the function name in parentheses when you make the call. For example, if you wanted to call only `memcpy` from the header file and use the built-in functions for other memory-related functions, code the function call as follows:

```
(memcpy)(buf1, buf2, len)
```

Note: When **NOOPT** or **COMPACT** is specified, the compiler might not expand all built-in functions.

The compiler can also generate inline code for some of the C library functions, if the appropriate header files are included in the source code. The inline code behaves exactly the same as these C library functions. For more information, see Using hardware built-in functions in .

The following table lists the C library built-in functions and the header files that they belong to.

Table 236. C-library built-in functions

Built-In Function	Header File
`abs()`	`stdlib.h`
`alloca()`	`stdlib.h`
`ceil()`“1” on page 166	`math.h`
`ceilf()`“1” on page 166	`math.h`
`ceill()`“1” on page 166	`math.h`
`decabs()`	`decimal.h`
`decchk()`	`decimal.h`
`decfix()`	`decimal.h`
`fabs()`“1” on page 166	`math.h`
`floor()`“1” on page 166	`math.h`
`floorf()`“1” on page 166	`math.h`
`floorl()`“1” on page 166	`math.h`
`fortrc()`	`stdlib.h`
`memchr()`	`string.h`
`memcpy()`	`string.h`
`memcmp()`	`string.h`
`memset()`	`string.h`
`strcat()`	`string.h`
`strchr()`	`string.h`
`strcmp()`	`string.h`
`strcpy()`	`string.h`

Table 236. C-library built-in functions (continued)

Built-In Function	Header File
`strlen()`	`string.h`
`strncat()`	`string.h`
`strncmp()`	`string.h`
`strncpy()`	`string.h`
`strrchr()`	`string.h`
`wmemchr()` "2" on page 166	`wchar.h`
`wmemcmp()` "2" on page 166	`wchar.h`
`wmemcpy()` "2" on page 166	`wchar.h`
`wmemset()` "2" on page 166	`wchar.h`

Notes:

1. The compiler only attempts to generate inline code for this built-in function when the **`OPTIMIZE(2)`** compiler option is in effect.
2. The compiler only attempts to generate inline code for this built-in function when the **`ARCH(7)`** compiler option is in effect. LP64 compiles will not generate inline code.

Related information

- For detailed information on how to use vector built-in functions to access and operate vector elements, see Chapter 3, "Using vector programming support," on page 27.

__builtin_expect

You can use the `__builtin_expect` built-in function to indicate that an expression is likely to evaluate to a specified value. The compiler can use this knowledge to direct optimizations. This built-in function is portable with the GNU C/C++ `__builtin_expect` function.

The prototype of this built-in function is as follows:

```
long __builtin_expect (long exp, long c);
```

where `exp` is the integral-type expression to be evaluated and `c` is the expected value of the expression.

If `exp` does not actually evaluate at run time to the predicted value `c`, performance might suffer. Therefore, you must use this built-in function with caution.

Platform-specific functions

The built-in functions in this section are related to C-library functions that are z/OS specific. The full description of each function can be found in .

Table 237. Platform-specific built-in functions

Built-In Function	Header File
`cds()`	`stdlib.h`
`cs()`	`stdlib.h`

Note: `cds()` and `cs()` are masking macros. The system header expands them to the `__cds` and `__cs`. It is advisable to use the hardware functions instead of the library functions whenever possible.

Examples

- You can use the following macros rather than their equivalent functions, if you include the `ctype.h` header file.

isalnum()	isalpha()	isblank()	iscntrl()	isdigit()
isgraph()	islower()	isprint()	ispunct()	isspace()
isupper()	isxdigit()	tolower()	toupper()	

- If you are using the `__cs1` or `__cds1` function with arguments other than the ones declared in the prototypes in `stdlib.h`, the compiler might not be able to generate correct code at OPT. In this case, use the NOANSIALIAS option.
- Typically, arrays are compared element-by-element, using a loop. When you compare two arrays for equality, replace the loop with the `memcmp()` library function. This could result in the execution of many machine instructions being replaced by the execution of a only a few machine instructions.

More efficient comparison with `amemcmp()` library function	**Less efficient comparison in a loop**
`if (!memcmp (a, b, sizeof(a)))` `  /*    arrays are equal    */`	`int a[1000], b[1000];` `for (i = 0; i < 1000; ++i)` `  if (a[i] != b[i])` `  break;` `if (i == 1000)` `  /*    arrays are equal     */`

- The C language does not allow structure comparison, because structures might contain padding bytes with undefined values. In cases where you know that no padding bytes exist, use `memcmp()` to compare structures. The AGGREGATE compiler option is used to obtain a structure and union map.
- The `memset()` library function should be used to initialize a character buffer and to initialize an array to a repetitive byte pattern (such as zeros).
- Use `memset()` to clear structs, unions, arrays or character buffers as follows:

```
char c[10];

for (i = 0; i < 10; i++)           /*  do not use  */
  c[i] = ' ';

memset (c, ' ', sizeof (c));       /*  better      */
```

- Use the `alloca()` function to automatically allocate memory from the stack. This function frees memory at the end of a function call when Enterprise Metal C for z/OS collapses the stack.
- When using `strlen()`, do not hide size information. Less code is needed for `strlen()` when the upper bound is known at compile time.

```
char    small_str_array[100];
char   *small_str_ptr;
:
x = strlen(small_str_ptr);   /*  unknown upper bound  */

x = strlen(small_str_array); /*  better  */
```

- When concatenating strings, use `strcat()`.
- When performing character-to-integer conversions, use `atoi()` rather than `sscanf()`.

- Whenever possible, replace `str`*xxx*`()` functions with their corresponding `mem`*xxx*`()` functions, because `mem`*xxx*`()` functions are more efficient. You can minimize the execution cost of a `str`*xxx*`()` function by using fixed-length character buffers to save the length of incoming strings (including null terminators) for subsequent calls to `memcpy()` and `memcmp()`.

```
total_len = strlen (s) + 1;
⋮
for (i = 0; i < 10; i++)
  if (memcmp (s, t[i], total_len) == 0) /* total_len ≤ sizeof(t) */
⋮

memcpy (a, s, total_len);
```

  If you try to replace all `strcmp()` calls with a `memcmp()` call taking a `strlen()` value of one of the strings, the result might be an attempt to access protected storage which follows the shorter string. Such an attempt could cause an exception because `memcmp()` does not stop comparing strings when it encounters a null in one of the strings.
- Whenever possible, replace `wcsxxx()` functions with their corresponding `wmemxxx()` functions, because `wmemxxx()` functions are more efficient. You can minimize the execution cost of a `wcsxxx()` function by using fixed-length wide character buffers to save the length of incoming wide character strings (including null terminators) for subsequent calls to `wmemcpy()` and `wmemcmp()`.

Chapter 6. Improving performance with compiler options

This information discusses and lists the Enterprise Metal C for z/OS compiler options that you can use to improve application performance.

Using the OPTIMIZE option

During optimization, the compiler changes the unoptimized code sequences, derived from the source code, into equivalent code sequences that execute faster and usually require less memory space. It is also possible for an expression that would normally cause an exception to be removed by optimization, thus preventing the exception.

Note: You can optimize code by specifying either OPTIMIZE(2) or OPTIMIZE(3). Optimized code takes significantly more time to compile than unoptimized code, but will likely result in faster-running code. There is no guarantee that the compile time at OPTIMIZE(3) will remain similar from release to release.

Because the optimization is achieved by transforming the code using knowledge obtained from a larger program context, the direct correspondence between source and object code is often lost. Optimized code is also more sensitive to subtle coding errors.

One example of a subtle coding error is to type cast a pointer variable incorrectly. The compiler assumes ISO conformance when doing optimization. If your program does not conform, you may receive undefined results. For more information, see “ANSI aliasing rules” on page 155 and “Using ANSI aliasing rules” on page 157.

Optimizations performed by the compiler

The compiler performs several optimizations, including:

Inlining
Inlining replaces certain function calls with the actual code of the function being performed. For more information on inlining, see “Inlining” on page 173.

For Enterprise Metal C for z/OS, automatic inlining is performed by default when you specify OPTIMIZE. You can override this inlining by using the NOINLINE option. For more information, see INLINE | NOINLINE in .

Value numbering
Value numbering involves local constant propagation, local expression elimination, and folding several instructions into a single instruction.

Straightening
Straightening is rearranging the program code to minimize branching logic and to combine physically separate blocks of code.

Common expression elimination
Common expressions recalculate the same value in a subsequent expression. The duplicate expression can be eliminated by using the previous value. This is done even for intermediate expressions within expressions.

If your program contains the following statements, the common expression `c + d` is saved from its first evaluation and is used in the subsequent statement to determine the value of `f`.

```
a = c + d;
    .
    .
    .
f = c + d + e;
```

Code motion
If variables used in a computation within a loop are not altered within the loop, it may be possible to perform the calculation outside of the loop and use the results within the loop.

Strength reduction
Less efficient instructions are replaced with more efficient ones. For example, in array addressing, an add instruction replaces a multiply.

Constant propagation
Constants used in an expression are combined and new ones generated. Some mode conversions are done, and compile-time evaluation of some intrinsic functions takes place.

Instruction scheduling
Instructions are reordered to minimize execution time.

Dead store elimination
The compiler eliminates stores when the value stored is never referred to again. For example, if two stores to the same location have no intervening load, the first store is unnecessary, and is therefore removed.

Dead code elimination
The compiler may eliminate code for calculations that are not required. Other optimization techniques may cause code to become dead.

Graph coloring register allocation
The compiler uses a global register allocation for the whole function, thereby allowing variables to be kept in registers rather than in memory.

These optimization techniques may be performed both locally and globally. Increases in storage and compile time requirements over NOOPT will occur. Higher levels of optimization may perform the same options more rigourously as well as adding additional options.

Aggressive optimizations with OPTIMIZE(3)

The compiler optimizes more aggressively with OPTIMIZE(3) than with OPTIMIZE(2). Code may be moved, and computations may be scheduled, even if this could potentially raise an exception.

OPTIMIZE(3) may place instructions onto execution paths where they will be executed when they may not have been according to the actual semantics of the program. For example, a loop-invariant floating-point computation that is found on some, but not all, paths through a loop will not be moved using OPTIMIZE(2) because the computation may cause an exception. For OPTIMIZE(3), the compiler will move the computation because it is not certain to cause an exception.

The same is true for moving loads. Although a load through a pointer is never moved, loads off the static or stack base register are considered movable using OPTIMIZE(3). Loads in general are not considered to be absolutely safe using OPTIMIZE(2) because a program can contain a declaration of a static array a of 10 elements and load a`[60000000003]`, which could cause a segmentation violation.

The same concepts apply to scheduling. In the following example, using OPTIMIZE(2), the computation of b+c is not moved out of the loop for two reasons:

- It is considered dangerous because it is a floating-point operation
- It does not occur on every path through the loop

```
...
int i;
float a[100], b, c;
for (i=0; i < 100; i++)
{
    if (a[i] < a[i+11])
    a[i] = b + c;
}
...
```

At OPTIMIZE(3), the computation b + c is moved out of the loop.

Some general differences with OPTIMIZE(2) are as follows:

- Increased optimization scope, typically to encompass a whole procedure
- Specialized optimizations that might not help all programs
- Optimizations that require large amounts of compile time or space
- Elimination of implicit memory usage
- Activation of NOSTRICT, which allows some reordering of floating-point computations and potential exceptions

Because OPTIMIZE(3) implies the NOSTRICT option, certain floating-point semantics of your application can be altered to gain execution speed. These typically involve precision trade-offs such as the following operations:

- Reordering of floating-point computations
- Reordering or elimination of possible exceptions (for example, division by zero or overflow)
- Combining multiple floating-point operations into single machine instructions; for example, replacing an add then multiply with a single more accurate and faster float-multiply-and-add instruction

You can still gain most of the benefits of OPTIMIZE(3) while preserving precise floating-point semantics by specifying STRICT. This is only necessary if a particular level of floating-point computational accuracy, as compared with NOOPT or OPTIMIZE(2) results, is important. You can also specify STRICT if your application is sensitive to floating-point exceptions, or if the order and manner in which floating-point arithmetic is evaluated is important. Largely, without STRICT, the difference in computed values on any one source-level operation is very small compared to lower optimization levels. However, the difference can compound if the operation involved is in a loop structure, and the difference becomes additive.

Optimization option levels

You can use the metalc utility options to specify five base optimization levels, which map to the z/OS batch options as follows:

- **-00** or NOOPT, almost no optimization, best for getting the most debugging information
- **-02** or OPTIMIZE(2), strong low-level optimization that benefits most programs
- **-03** or OPTIMIZE(3), intense low-level optimization analysis
- **-04** or OPTIMIZE(3), HOT, IPA(LEVEL(1)), all of **-03** plus detailed loop analysis and basic whole-program analysis at link time
- **-05** or OPTIMIZE(3), HOT, IPA(LEVEL(2)), all of **-04** and detailed whole-program analysis at link time

Note: -01 level is not supported.

Optimization progression

Table 238 on page 171 details options you should use with each level and some useful additional options.

Table 238. Optimization levels and options

The metalc utility optimization option level	Additional batch options implied by optimization level	Additional recommended batch options
-00	None	ARCH(n)
-02	None	ARCH (n) INLINE (to tune inlining) TUNE(n)
-03	NOSTRICT	ARCH(n) TUNE(n)

Table 238. Optimization levels and options (continued)		
The metalc utility optimization option level	**Additional batch options implied by optimization level**	**Additional recommended batch options**
-O4	All of OPTIMIZE(3) plus: HOT IPA(LEVEL(1))	ARCH(n) TUNE(n) PDF
-O5	All of OPTIMIZE(3) plus: HOT IPA(LEVEL(2))	ARCH(n) TUNE(n) PDF

While Table 238 on page 171 provides a list of the most common compiler options for optimization, the compiler offers optimization facilities for almost any application. For more information, see "Additional options that affect performance" on page 180.

Processor optimization capabilities with ARCH and TUNE options

ARCHITECTURE option

The ARCHITECTURE option specifies the architectural level for which the executable program's instructions will be generated.

The ARCHITECTURE option instructs the compiler to structure your application to execute on a particular set of machines that support the specified instruction set and later. The choice of processor gives you the flexibility of compiling your application to execute optimally on a particular machine or on any higher-level architecture machines but still have as much architecture-specific optimization applied as possible.

Using the correct ARCHITECTURE option is the most important step in influencing chip-level optimization. The compiler uses the ARCHITECTURE option to make both high and low-level optimization decisions and trade-offs. The ARCHITECTURE option allows the compiler to access the full range of processor hardware instructions and capabilities when making code generation decisions. Even at low optimization levels, specifying the correct target architecture can have a positive impact on performance.

For example, to compile applications with the Enterprise Metal C for z/OS compiler to produce code that uses instructions available on the z13® models, use ARCHITECTURE(11).

TUNE option

The TUNE option specifies for which architectural level the executable program will be optimized. The TUNE option allows the compiler to take advantage of differences (such as scheduling of instructions) in architectural levels.

Use the TUNE option to direct the optimizer to bias optimization decisions for executing the application on a particular architecture but not preventing the application from running on other architectures. The default TUNE setting depends on the setting of the ARCHITECTURE option. If the ARCHITECTURE option selects a particular machine architecture, the range of TUNE suboptions that are supported is limited by the chosen architecture and all architectures above that level. Using TUNE allows the optimizer to perform transformations, such as instruction scheduling, so that resulting code executes most efficiently on your chosen TUNE architecture.

Use TUNE to specify the most common or important processor where your application executes. For example, if your application usually executes on zEC12 models but sometimes executes on z13 models, use TUNE(10). The code generated executes more efficiently on zEC12 models but can run correctly on z13 models.

Inlining

Inlining replaces certain function calls with the actual code of the function and is performed before all other optimizations. Not only does inlining eliminate the linkage overhead, it also exposes the entire called function to the caller, which enables the compiler to better optimize your code.

Note: See "Inlining under IPA" on page 174 for information on differences in inlining under IPA.

The following types of calls are not inlined:

- A call where the number of parameters on the call does not match that on the function definition. An example of this is a variable argument function call.
- A call that is directly recursive; the routine calls itself.
- K&R style `var_arg` functions.

Selectively marking code to inline

The Enterprise Metal C for z/OS inliner supports two modes of running: selective and automatic.

Selective mode enables you to specify, in your source code, the functions that you do, and do not, want inlined.

If you know which functions are frequently invoked from within a compilation unit, you can mark them for inlining,

- Add the appropriate **`#pragma inline`** directives in your source and compile with INLINE.
- You can also use the `always_inline` function attribute to inline a function, regardless of whether optimization was specified at compile time.

If your code contains complex macros, the macros can be made into static routines that are marked to be inlined at no execution-time cost. All static routines that are interfaces to a data object can be placed in a header file.

Automatically choosing functions to inline

Automatic mode assists you with starting to optimize your code. It allows the compiler to choose potential functions to inline. The compiler will inline all routines that are less than the *threshold* in abstract code units (ACUs) until the function that the functions are inlined into is greater than *limit* abstract code units. The *threshold* and *limit* parameters are defined as follows:

threshold
: Maximum relative size of a function to inline. The default value is 100 Abstract Code Units (ACUs). ACUs are proportional in size to the executable code in the function; your code is translated into ACUs by the compiler. Specifying a threshold of `0` is equivalent to specifying NOAUTO. Note that the proportion of ACUs to executable code in a function is different under IPA.

limit
: Maximum relative size a function can grow before auto-inlining stops. The default is 1000 ACUs for the specific function. Specifying a limit of `0` is equivalent to specifying NOAUTO.

Note: When functions become too large, runtime performance can degrade.

Note: Inlining debugging functions or functions that are rarely invoked can degrade performance. Use the **`#pragma noinline`** directive to instruct the automatic inliner to not inline these types of functions. The **`#pragma inline`** and the **`#pragma noinline`** directives and the `inline` keyword are honored by automatic inlining regardless of the *limit* and *threshold* you have specified. For more information, see #pragma inline / noinline in .

Modifying automatic inlining choices

While automatic inlining is the best choice the compiler can make for you, you can further improve your performance. Use **`#pragma inline`** and **`#pragma noinline`** to reduce the need to modify your inlining

choices when you change your application. You may want to wait until you have a stable application before you do the following steps.

1. Add **#pragma noinline** to your source to insure that functions, such as routines for debugging or handling exceptions, do not get inlined.
2. Add **#pragma inline** directive to any frequently used routines to ensure that it gets inlined.
3. You should also vary the limit and threshold values.
   - The inline report tells you the abstract code units (ACUs) for each function. These should help you determine an appropriate *threshold* to start from. In general, your initial *threshold* should be as small as possible, and your initial limit should be in the 1000 to 2000 range.
   - Increase the *threshold* by an increment small enough to catch a few more routines each time.
   - Change the limit when you wish. Because performance will improve as a function of both the `limit` and the `threshold` values, it is not recommended that you change both `limit` and `threshold` at the same time.
4. Repeat the process until you feel that you have found the best performance parameters. You should run your application to determine if the tuning has found the best performance parameters.
5. When you are satisfied with the selection of inlined routines, add the appropriate **#pragma inline** directives or `inline` keywords to the source. That is, when the selected routines are forced with these directives, you can then compile the program in selective mode. This way, you do not need to be affected by changes made to the heuristics used in the automatic inliner.

Overriding inlining defaults

Automatic and selective inlining are performed when the OPTIMIZE compiler option is specified. You can override this by specifying the NOINLINE option when you specify your optimization level. You can also override this by specifying the **#pragma noinline** directive for a particular function. For more information, see #pragma inline / noinline in .

Inlining under IPA

The IPA Inliner functions differently from the regular inliner:

- It performs inlining across compilation units, rather than within a compilation unit.
- It handles inlining of functions with variable argument lists.
- It inlines calls from recursive cycles (for example, where function A calls function B calls function C calls function A). However, it avoids making the functions too large.

For more information about IPA, see “Using the IPA option” on page 175.

Using the HOT option

The HOT option enables the compiler to request high-order transformations on loops during optimization, which gives you the ability to generate more highly optimized code.

Loops typically account for the majority of the execution time of most applications and the HOT optimizer performs in-depth analysis of loops to minimize their execution time. Loop optimization techniques include: interchange, fusion, unrolling of loop nests, and reducing the use of temporary arrays. There are three goals in these optimizations:

- Reducing the costs of memory access through the effective use of caches and translation look-aside buffers (TLBs). Increasing memory locality reduces cache/TLB misses.
- Overlapping computation and memory access through effective utilization of the data prefetching capabilities provided by the hardware.
- Improving the utilization of processor resources through reordering and balancing the usage of instructions with complementary resource requirements. Loop computation balance typically involves load/store operations balanced against floating-point computations.

Using the IPA option

Interprocedural Analysis (IPA), through the IPA option, can also improve the execution time of your application. IPA is a mechanism for performing optimizations across compilation unit boundaries. It also performs optimizations not otherwise available with the Enterprise Metal C for z/OS compiler, such as:

- Inlining across compilation units
- Program partitioning
- Coalescing of global variables
- Code straightening
- Unreachable code elimination
- Call graph pruning of unreachable functions

This information provides an overview of the Interprocedural Analysis (IPA) processing that is available through the IPA compiler option. For more information, see:

- For the effects of IPA on compiling, compiler options, and compiler listings: IPA considerations in
- For the effects of IPA on pragmas: IPA effects in

Types of procedural analysis

The Enterprise Metal C for z/OS compiler performs both intraprocedural and interprocedural analysis.

Intraprocedural analysis is a mechanism for performing optimization for each function in a compilation unit, using only the information available for that function and compilation unit.

Interprocedural analysis is a mechanism for performing optimization across function and compilation unit boundaries. When inlining is in effect, the compiler performs a limited form of interprocedural analysis, where it only applies within a compilation unit.

Interprocedural analysis through the IPA compiler option improves upon the limited interprocedural analysis described above. When you invoke interprocedural analysis through the IPA option, the compiler performs optimizations across the entire program. It also performs optimizations not otherwise available with the compiler. The types of optimizations performed include:

Inlining across compilation units
Inlining replaces certain function calls with the actual code of the function. Inlining not only eliminates the linkage overhead but also exposes the entire function to the caller and thus enables the compiler to better optimize your code.

Program partitioning
Program partitioning improves performance by reordering functions to exploit locality of reference. Functions that call each other frequently will be closer together in memory.

Coalescing of global variables
The compiler puts global variables into one or more structures and accesses the variables by calculating the offsets from the beginning of the structures. This lowers the cost of variable access and exploits data locality.

Code straightening
Code straightening streamlines the flow of your program.

Unreachable code elimination
Unreachable code elimination removes unreachable code within a function.

Call graph pruning of unreachable functions
Call graph pruning of unreachable functions removes code that is 100% inlined or never referenced.

Intraprocedural constant and set propagation
IPA propagates floating point and integer constants to their uses and computes constant expressions at compile time. Also, variable uses that are known to be one of several constants can result in the folding of conditionals and switches.

Intraprocedural pointer alias analysis
IPA tracks pointer definitions to their uses, resulting in more refined information about memory locations that a pointer dereference may use or define. This enables other parts of the compiler to better optimize code around such dereferences. IPA tracks data and function pointer definitions. When a pointer dereference can only refer to a single memory location or function, the dereference is rewritten to be an explicit reference to the memory location or function.

Intraprocedural copy propagation
IPA propagates expressions defining some variables to the uses of the variable. This creates additional opportunities for constant expression folding. It also eliminates redundant variable copies.

Intraprocedural unreachable code and store elimination
IPA removes definitions of variables that cannot be reached, along with the computation feeding the definition.

Conversion of reference (address) arguments to value arguments
IPA converts reference (address) arguments to value arguments when the formal parameter is not written in the called procedure.

Conversion of static variables to automatic (stack) variables
IPA converts static variables to automatic (stack) variables when their use is limited to a single procedure invocation.

The execution time for code optimized using interprocedural analysis (IPA compile and link) is normally faster than for code optimized using intraprocedural analysis (IPA compile only) or the OPT compiler option. Please note that not all applications are suited for IPA optimization and the performance gains realized from using IPA will vary.

Note: For additional information about using the IPA(LINK) option, see "IPA(LINK) option and exploitation of 64-bit virtual memory" on page 8.

Compiler processing flow

IPA changes the flow of compiler processing. The following sections explain the differences.

Regular compiler execution

If you specify the NOIPA compiler option (the default), the compiler processes source files, as shown in Figure 24 on page 176. The output is the HLASM source code for each source file processed.

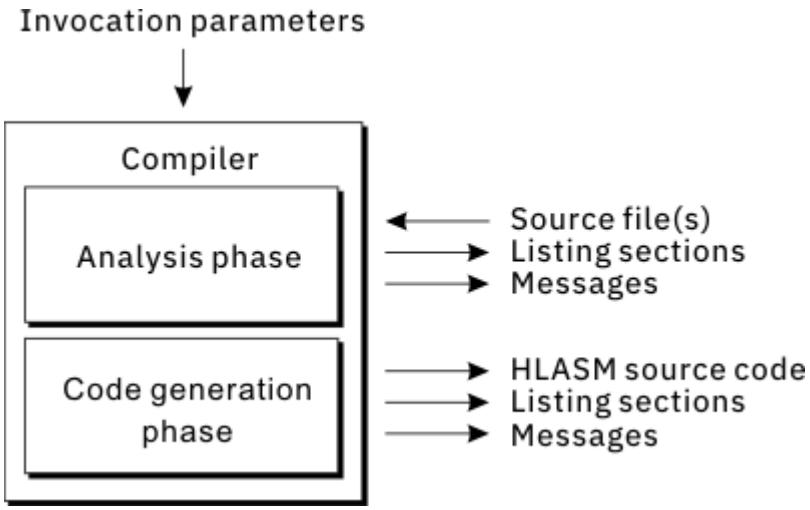


Figure 24. Flow of regular compiler processing

Compiler execution with IPA

IPA processing consists of two steps: IPA compile and IPA link. You run the IPA compile step once for each compilation unit, and run the IPA link step once for the program as a whole. The final output is a

single IPA-optimized object module which you must bind with the binder to produce an executable load module.

Note: If you want to get the maximum benefit from IPA, run both the IPA compile and IPA link steps.

You can invoke the IPA compile step in the same environments that you use for a regular compilation.

IPA compile step processing

You invoke the IPA compile step by specifying the IPA(NOLINK) compiler option, as shown in Figure 25 on page 177. (NOLINK is the default suboption). During the IPA compile step, the compiler creates optimized objects. These objects contain information that the IPA link step can use for further optimization.

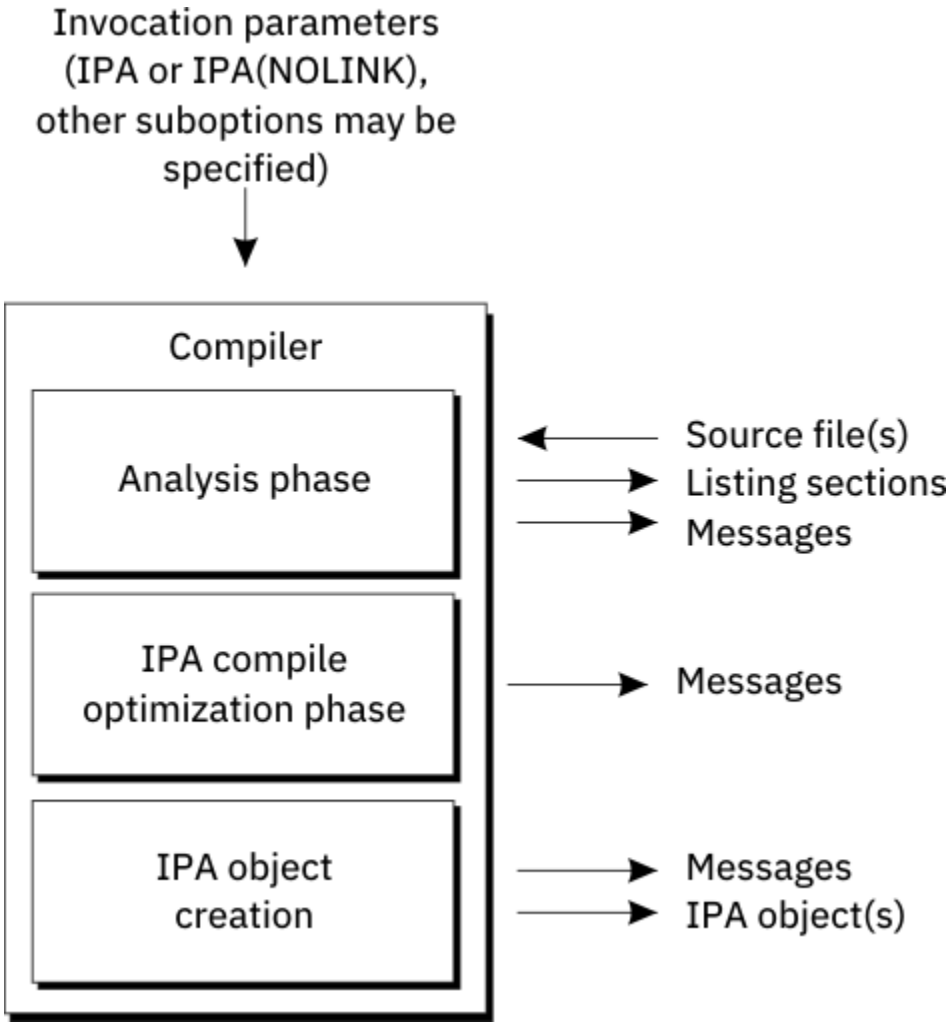


Figure 25. IPA compile step processing

The following processing takes place for each compilation unit that you specify for the IPA compile step:

1. The compiler determines the final suboptions for the IPA option, based upon the compiler options and IPA suboptions that you specified. This is necessary because the compiler does not support some combinations of compiler options and IPA suboptions. The compiler issues a warning message if it finds unsupported combinations.
2. The compiler promotes some IPA suboptions based upon the presence of related compiler options and issues informational messages if it does so. For more information, see IPA considerations in .
3. The compiler generates an IPA object file. This object file contains control information for a compilation unit required for the IPA link step.

   Each IPA object contains a CSECT that includes the ESD name `@@IPAOBJ`.

IPA link step processing

You invoke the IPA link step by specifying the IPA(LINK) compiler option, as shown in Figure 26 on page 178. During this step, the compiler links the IPA objects that were produced by the IPA compile step (along with non-IPA object files and load modules, if specified), does partitioning, performs optimizations, and generates the final object code.

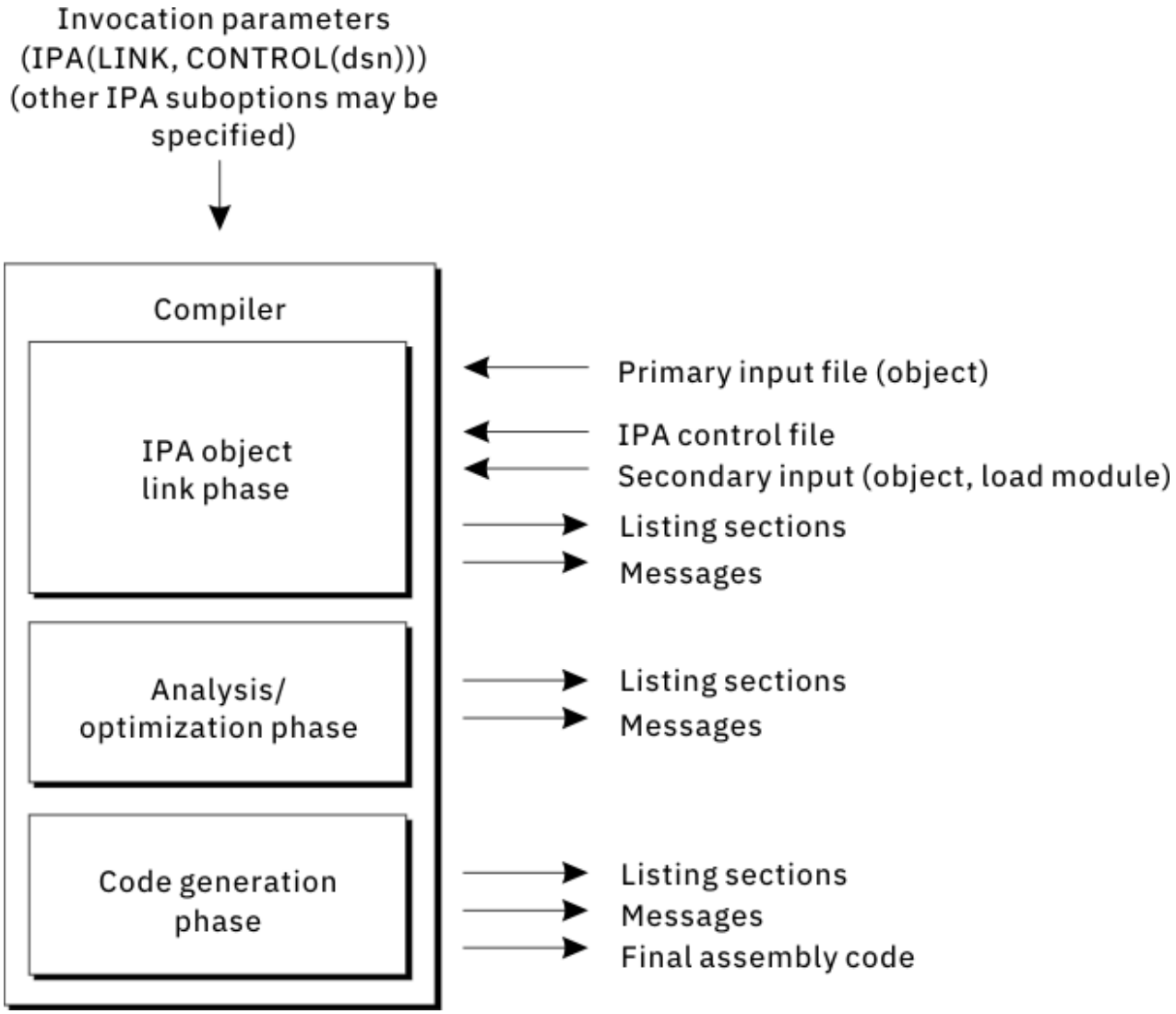


Figure 26. IPA link step processing

The following processing takes place:

1. The compiler determines the final suboptions for the IPA option, based upon the compiler options and IPA suboptions you specify. This is necessary because some combinations of compiler options and IPA suboptions are unsupported. The compiler issues informational and warning messages for unsupported combinations.
2. The compiler links IPA object files, as well as non-IPA object files and load modules (if specified). The compiler also merges information from the IPA compile step.

   Input for the link step comes from one of three sources:

   - The primary input file (specified by the SYSIN ddname). This can be either:
     – A set of IPA link control statements that you create

These may be INCLUDE and LIBRARY IPA link control statements that explicitly identify secondary input files. IPA uses the same control statement format (with some exceptions) used by the binder.

- The IPA object file from the compilation unit that contains the `main` function or fetchable entry point. If you specify this file, the compiler searches for all other IPA files using the SYSLIB ddname.

- One or more secondary input files

  The secondary input file may contain:

  - IPA object files or PDS libraries
  - Conventional object files or PDS libraries
  - Load module libraries
  - z/OS UNIX archive libraries
  - IPA link control statements

  These secondary input files are to be used for autocall searches. You can specify these files through the SYSLIB ddname or explicitly include them through INCLUDE or LIBRARY IPA link control statements on the IPA link step.

  The IPA link step resolves external references using explicit and autocall resolution. This allows IPA to identify the static and global data and the external references for the whole program.

  Ensure that you do not accidentally specify FB, LRECL 80 source files as input to the IPA link step. The IPA link step will assume that records from these files contain valid object information, and will retain them in the object file. When the linkage editor processes the object file, it will determine the records to be invalid, and will issue diagnostic messages.

- The IPA link step control file. This file contains additional IPA control directives. The CONTROL suboption of the IPA compiler option identifies this file. For more information, see IPA Link step control file in .

3. As objects are processed, IPA link step builds the program call graph, merging the IPA object code according to its place in the call graph. If necessary, IPA link step stores non-IPA object code for inclusion in the final object file, and converts load module library members into object format for inclusion in the final object file.
4. The compiler performs optimizations across the call graph. You specify the type and extent of optimizations using the LEVEL suboption of the IPA compiler option.
5. IPA link step divides the program call graph into separate units called partitions. Partitioning of the call graph is controlled by:

   - The partition size limit that is specified in the IPA control file.
   - The connectivity of your program. IPA places code that is isolated from the rest of the program into a separate partition.
   - Resolution of conflicting effects between the compiler options and pragmas specified for compilation units processed during the IPA Compile step. These are the compiler options and pragmas that generate information during the analysis phase of the compiler for input to the code-generation phase.

   IPA link step produces a final single object module for the program from these partitions.

You must bind the IPA single object module to produce the executable module. You need to add the assembly step to produce the object file from the IPA link generated HLASM source file. You also need to supply the object file produced by the assembler along with all other library data sets to the binder for producing the final executable program.

Notes:

- An object file produced by an IPA compile that contains IPA object or combined IPA and conventional object information can be used as input to the IPA link of the same or later Version/Release.

- An object file produced by an IPA compile that contains IPA object or combined IPA and conventional object information cannot be used as input by the IPA link of an earlier Version/Release. If this is attempted, the IPA link will issue an error diagnostic message.
- If the IPA object is recompiled by a later IPA compile, additional optimizations may be performed and the resulting application program may perform better.

Additional options that affect performance

The following topics describe compiler options that affect performance. For more information, see Compiler options in .

AGGRCOPY

The AGGRCOPY option specifies whether aggregate assignments might have overlapping source and target locations. AGGRCOPY(NOOVERL), which is the default, asserts to the compiler that the source and destination for structure and union assignments do not overlap. This assumption enables destructive copy operations for structures and unions, which can improve performance.

ANSIALIAS

The ANSIALIAS option specifies whether type-based aliasing is to be used during optimization. Type-based aliasing will improve optimization. For more information about ANSI aliasing, see “ANSI aliasing rules” on page 155 and “Using ANSI aliasing rules” on page 157.

ASSERT(RESTRICT)

The ASSERT(RESTRICT) option enables optimizations for restrict qualified pointers.

COMPACT

When the COMPACT option is in effect, the compiler favors optimizations that tend to limit the growth of the code. Depending on your specific program, the object size may increase or decrease and the execution time may increase or decrease. Any time you change your program, or change the release of the compiler, you should re-evaluate your use of the COMPACT option.

COMPRESS

Use the COMPRESS option to suppress the generation of function names in the function control block to reduce the size of your application's load module. The amount of reduction depends on the average function size in the application, as compared to the length of the function name.

FLOAT

Some of the FLOAT suboption provide precise control over the handling of floating-point calculations with binary floating-point numbers. Some of the most frequently used suboptions that affect performance when used with FLOAT(IEEE) are listed as follows:

FOLD
Enables compile time evaluation of floating-point calculations. You should disable folding only if your application must handle certain floating-point exceptions such as overflow or inexact. FLOAT(FOLD) is the default.

MAF
Makes floating-point calculations faster and more accurate by using floating-point multiply-add instructions where appropriate. However, the results might not be equivalent to those from similar calculations performed at compile time or on other types of computers.

NORRM
Allows the compiler to assume that the rounding mode is always round-to-nearest. FLOAT(NORRM) is the default.

HGPR

The HGPR option enables the compiler to exploit 64-bit General Purpose Registers (GPRs) in 32-bit programs targeting z/Architecture hardware.

LIBANSI

The LIBANSI option specifies whether or not all functions with the name of an ANSI C library function are in fact the ANSI functions. This allows the compiler to generate code based on existing knowledge concerning the behavior of the function. For example, the compiler will determine whether any side effects are associated with a particular library function.

PREFETCH

The PREFETCH option inserts prefetch instructions automatically where there are opportunities to improve code performance.

RESTRICT

The RESTRICT option indicates to the compiler that all pointer parameters in some or all functions are disjoint.

ROCONST

The ROCONST option specifies that the `const` qualifier is respected by the program. Variables that are defined with the `const` keyword are not overridden by a casting operation.

ROSTRING

The ROSTRING option specifies that strings are placed in read-only memory. It has the same effect as the **`#pragma strings(readonly)`** directive.

STRICT

The STRICT option prevents optimizations done by default at optimization levels OPT(3), and optionally at OPT(2), from re-ordering instructions that could introduce rounding errors.

STRICT_INDUCTION

With strict induction, induction (loop counter) variables are not optimized. This guards against problems that can occur if an optimized induction variable overflows.

If it is certain that the induction variables will not overflow, use the NOSTRICT_INDUCTION option. This option can improve the performance of induction variables that are smaller than the register size on the processor.

UNROLL

The UNROLL option gives the user the ability to control the amount of loop unrolling done by the compiler. Loop unrolling exposes instruction level parallelism for instruction scheduling and software pipelining and thus can improve a program's performance. It should be used in conjunction with #pragma unroll.

VECTOR

The VECTOR option controls whether the compiler enables the vector programming support and automatically takes advantage of vector/SIMD instructions. The VECTOR option provides potential performance improvements in the following cases:

Binary floating-point data types: float and long double
When the VECTOR option is specified with ARCH(12) and FLOAT(IEEE), the `long double` and `float` data types can be processed in the vector registers.

Binary floating-point data type: double
When the VECTOR option is specified with ARCH(11) or higher and FLOAT(IEEE), the `double` data types can be processed in the vector registers.

Built-in library functions
When the VECTOR option is specified with ARCH(11) or higher, certain built-in library functions can take advantage of vector string instructions to accelerate the processing of strings of character data.

SIMD instructions
When the AUTOSIMD suboption is in effect, the compiler generates code, when possible, using the SIMD instructions. SIMD instructions calculate several results at one time, which is faster than calculating each result sequentially.

Related information

- VECTOR | NOVECTOR in .

Chapter 7. Balancing compilation time and application performance

Compilation time increases as the level of optimization increases. An end user requires that an application run as fast as possible, and therefore will compile with the maximum optimization possible. Conversely, a developer rebuilds an application many times while debugging a problem, and therefore will compile with the minimum optimization possible. In addition, a developer might need to implement debugging tools, or activate extra debugging code, both of which would affect the performance of the application. This information discusses how to determine the proper balance between compilation time and application performance.

General tips

The following list contains suggestions to support your efforts to debug programs, and reduce compilation time, and improve application performance.

- All builds for testing or production should be compiled with the optimization level at which you intend to ship the final product.
- Even if you compile with OPT(0) and debug on a regular basis, you should also do some testing at higher optimization levels to ensure that no aliasing rules or ANSI rules have been broken, which would cause the code to be optimized incorrectly.
- You can ensure the cleanest possible optimized compilations, as well as reduce the number of bugs that occur only at high optimization levels, by reviewing every warning issued by the compiler.

  Note: Warnings are often a sign that the compiler is not sure how to interpret the code. If the compiler is not sure how to interpret code at OPT(0), the code could cause an error at higher optimization levels or contribute to longer compilation times.
- The simpler the code is, the more easily the compiler can understand it and the faster it will compile. For more information, see Chapter 4, “Improving program performance,” on page 155.
- Generate production builds each week throughout the project cycle. This makes it easier to determine when problems entered the code base. Waiting until the end of a cycle to generate a build with high optimization can make it more difficult to find errors caused by coding that does not confirm to ANSI aliasing rules.
- Set up a build so that you can customize options for any source file, if necessary. For example, use a makefile for a UNIX System Services-based build with a default rule for compilation. You can then customize targets for source files that require different options. Similarly, use the OPTFILE compiler option for a JCL-based build. A build script can then use a project-level option file for all source files in a module. You can specify either of the following:
  - Both a project-level option file and additional specific options for a source file
  - A source-specific option file in the option list that follows the options file name
- Set up build scripts so that they can be used for both development and production builds to:
  - Eliminate a common source of errors (because it is necessary to update only one build environment)
  - Make it easier to reproduce and debug problems that occur only in the development build
  - Minimize occurrences of bugs that are reproducible only in the development build

Programmer tips

- You can add code to the beginning and end of a header file to ensure that it is not processed unnecessarily during compilation.

The following example contains code that is included in a header file called `myheader`.

```
#ifndef __myheader
    #ifdef __COMPILER_VER__
        #pragma filetag ("IBM-1047")
    #endif
    #define __myheader 1
    /* header file contents */
#endif
```

You must ensure that the `filetag` statement, if used, appears before the first statement or directive (except for any conditional compilation directives). The `ifndef` statement is the first non-comment statement in the header file (the actual token used after the `ifndef` statement is your choice). The `define` statement must follow; it cannot appear before the `filetag` statement, but it must appear before any other preprocessor statement (other than comments).

Note that the header can contain comment statements in any location. Using this format of header-file blocking will improve compilation time for programs where a header file is included more than once.

- Use the system header files from UNIX file system instead of partitioned data sets to improve compilation time. Specify the following compiler options to do this:

```
NOSEARCH SEARCH('/usr/include/metal/')
```

- With the MEMORY compiler option (the default), the compiler uses a hiperspace or memory file in place of a work file (if possible). This option increases compilation speed, but you might require additional memory to use it. If the compilation fails because of a storage error, either increase your storage size or recompile your program using the NOMEMORY option.
- If you want to improve your OPT compilation time at the expense of runtime performance, you can specify:

  MAXMEM
  Limits the amount of memory used for local tables of specific memory intensive optimizations. If this amount of memory is insufficient for a particular optimization, the compiler performs somewhat poorer optimization and issues a warning message. Reducing the MAXMEM value from 2G to 10M may disable some optimizations, which may cause some decrease in execution performance.

  NOINLINE
  Disables inlining, which might decrease the compilation time. There might also be a corresponding increase in execution time.

System programmer tips

- If you do a lot of application development on your machine, put the compiler and runtime library in the LPA. Similarly, if you are working in z/OS UNIX System Services also put the metalc utility in the dynamic LPA, LPA, or linklist.
- Use packs that are cached with DASD fast write.

  If you are working in z/OS UNIX System Services, give each user a separate mountable file system to avoid I/O contention.

  If the compiler is not in LPA, tune your jobs to avoid channel and pack contention when the headers and the compiler are on the same pack and multiple compile jobs are executing.
- You can define `/tmp` as a RAM disk by specifying:

```
FILESYSTYPE TYPE(TFS) ENTRYPOINT(BPXTFS)
```

This is described in more detail in .

Appendix A. Packaging considerations

When you develop a program, library, or application that will be shipped as a product, you should use SMP/E to manage the installation. This information provides hints and tips for packaging a C application. It assumes that you are familiar with SMP/E concepts and terminology. For more information about SMP/E and packaging rules, refer to the following manuals:

-
-
- *Standard packaging rules for MVS-based products*

The way you package your product may have a significant impact on its relationship with other products, its dependency on libraries, and the way it is eventually serviced. For this reason, you should make a packaging plan as part of the design process for your product.

For more information about these compiler options, see .

Compiler options

The CSECT option is useful when you compile a program that will be packaged as a product. You can use the CSECT compiler option or `#pragma csect` to assign names to CSECTs. This provides you with more control and flexibility when you service the product.

For more information about these compiler options, see .

Libraries

Your product can use various type of libraries:

Your own libraries
If your program uses your own libraries, you can statically bind the libraries with the program and consider them an integral part of the product.

Third-party libraries
If your application uses third-part vendor libraries, you should consider whether the linking is static or dynamic (if it is a DLL), and whether the libaries are upward-compatible. If you statically link a library with your application, you can use either the ++MOD method or the ++PROGRAM method, as described in “Linking” on page 185.

Linking

There are two ways to ship an application that is statically linked to a library:

- You can use the ++MOD command to build the application, and not perform the final link to the library until the product is installed. If the customer later installs a PTF for this library, your application will automatically be relinked.
- You can build the application and link it to the library, and then install it using the ++PROGRAM command. If a PTF is issued for the library, this will have no effect until you include the updated library in a PTF for your product.

++MOD method

If you want to do the final link-edit step during installation, use the ++MOD command statement in the MCS. You must compile and then partially link your program with any libraries that will not exist on the

customer's system, and then produce output in link-edited format. Any references to libraries that will exist on the customer's system are unresolved. Ship this link-edited module on the SMP/E tape.

At installation time, the application is linked to the libraries on the customer's system.

SMP/E supports the automatic library call facility through the use of SYSLIB DD statements. This allows you to implicitly include modules without explicitly specifying them in the JCLIN. This can provide flexibility if the link-edit structure of the application must change during servicing, for example because new functions are used.

When you service a ++MOD, you must ship your fixes using a ++PTF command statement. The SMP/E tape must contain the text deck (object files) in fixed-block 80 format. SMP/E invokes the link-editor to rebind the new text deck with the existing load module. You must name all of the CSECTs, using the CSECT compiler option or `#pragma csect`. (If you do not name the CSECTs, CSECT replacement would not happen. Old text records would accumulate in the load module as you ship out subsequent fixes for your product.)

To allow rebind, you must also use the EDIT=YES option in the bind step. This is the default.

++PROGRAM method

You can choose to do the final link step as part of your product build, and ship the output load module to your customer. The advantage is that the whole build process is under your control, and you can perform the final testing of the load module in your own controlled environment.

If service is applied to any linked library, this will have no effect on your product until you include the service in a PTF.

The ++PTF command, which is used for shipping and applying fixes, expects input in fixed-block 80 format. The output of the link step is not in this format. You can convert it as follows:

1. UNLOAD - use IEBCOPY to copy the module and its alias (if any) to a sequential file.
2. Run the SMP/E utility GIMDTS to convert the sequential file to a fixed-block 80 file.

Conceptually, the ++PROGRAM copies the whole load module to your customer's target dataset with no additional processing. You customer receives the module exactly as you ship it.

Appendix B. Accessibility

Accessible publications for this product are offered through IBM Knowledge Center (www.ibm.com/support/knowledgecenter/SSLTBW/welcome).

If you experience difficulty with the accessibility of any z/OS information, send a detailed message to the Contact z/OS web page (www.ibm.com/systems/z/os/zos/webqs.html) or use the following mailing address.

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
United States

Accessibility features

Accessibility features help users who have physical disabilities such as restricted mobility or limited vision use software products successfully. The accessibility features in z/OS can help users do the following tasks:

- Run assistive technology such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using the keyboard.
- Customize display attributes such as color, contrast, and font size.

Consult assistive technologies

Assistive technology products such as screen readers function with the user interfaces found in z/OS. Consult the product information for the specific assistive technology product that is used to access z/OS interfaces.

Keyboard navigation of the user interface

You can access z/OS user interfaces with TSO/E or ISPF. The following information describes how to use TSO/E and ISPF, including the use of keyboard shortcuts and function keys (PF keys). Each guide includes the default settings for the PF keys.

-
-
-

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users who access IBM Knowledge Center with a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line because they are considered a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that the screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol is placed next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol to provide information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, it indicates a reference that is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you must refer to separate syntax fragment OP1.

The following symbols are used next to the dotted decimal numbers.

? indicates an optional syntax element
The question mark (?) symbol indicates an optional syntax element. A dotted decimal number followed by the question mark symbol (?) indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that the syntax elements NOTIFY and UPDATE are optional. That is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! indicates a default syntax element
The exclamation mark (!) symbol indicates a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the dotted decimal number can specify the ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In the example, if you include the FILE keyword, but do not specify an option, the default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, the default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP applies only to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

*** indicates an optional syntax element that is repeatable**
The asterisk or glyph (*) symbol indicates a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area.

If you hear the lines `3* , 3 HOST, 3 STATE`, you know that you can include `HOST, STATE`, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write `HOST STATE`, but you cannot write `HOST HOST`.
3. The * symbol is equivalent to a loopback line in a railroad syntax diagram.

+ indicates a syntax element that must be included

The plus (+) symbol indicates a syntax element that must be included at least once. A dotted decimal number followed by the + symbol indicates that the syntax element must be included one or more times. That is, it must be included at least once and can be repeated. For example, if you hear the line `6.1+ data area`, you must include at least one data area. If you hear the lines `2+, 2 HOST,` and `2 STATE`, you know that you must include `HOST, STATE,` or both. Similar to the * symbol, the + symbol can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loopback line in a railroad syntax diagram.

Glossary

This glossary defines technical terms and abbreviations that are used in Enterprise Metal C for z/OS documentation. If you do not find the term you are looking for, refer to the index of the appropriate Enterprise Metal C for z/OS manual or view the IBM Glossary of Computing Terms (www.ibm.com/software/globalization/terminology).

The following cross-references are used in this glossary:

- See refers you from a term to a preferred synonym, or from an acronym or abbreviation to the defined full form.
- See also refers you to a related or contrasting term.

To view glossaries for other IBM products, go to IBM Glossary of Computing Terms (www.ibm.com/software/globalization/terminology).

A

abstract data type
A mathematical model that includes a structure for storing data and operations that can be performed on that data. Common abstract data types include sets, trees, and heaps.

access mode

1. The manner in which files are referred to by a computer. See also dynamic access, sequential access.
2. A form of access permitted for a file.

access specifier
A specifier that defines whether a class member is accessible in an expression or declaration. The three access specifiers are public, private, and protected.

addressing mode (AMODE)
The attribute of a program module that identifies the addressing range in which the program entry point can receive control.

address space
The range of addresses available to a computer program or process. Address space can refer to physical storage, virtual storage, or both.

aggregate

1. A structured collection of data objects that form a data type.

alert

1. A message or other indication that signals an event or an impending event.
2. To cause the user's terminal to give some audible or visual indication that an error or some other event has occurred.

alert character
A character that in the output stream causes a terminal to alert its user by way of a visual or audible notification. The alert character is the character designated by a '\a' in the C language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the alert function.

alias

1. An alternative name for an integrated catalog facility (ICF) user catalog, a file that is not a Virtual Storage Access Method (VSAM) file, or a member of a partitioned data set (PDS) or a partitioned data set extended (PDSE).

2. An alternative name used instead of a primary name.

aliasing
A compilation process that attempts to determine what aliases exist, so that optimization does not result in incorrect program results.

alignment
The storing of data in relation to certain machine-dependent boundaries.

alternate code point
A syntactic code point that permits a substitute code point to be used. For example, the left brace ({) can be represented by X'B0' and also by X'C0'.

American National Standards Institute (ANSI)
A private, nonprofit organization whose membership includes private companies, U.S. government agencies, and professional, technical, trade, labor, and consumer organizations. ANSI coordinates the development of voluntary consensus standards in the U.S.

American Standard Code for Information Interchange (ASCII)
A standard code used for information exchange among data processing systems, data communication systems, and associated equipment. ASCII uses a coded character set consisting of 7-bit coded characters. See also Extended Binary Coded Decimal Interchange Code.

AMODE
See addressing mode.

angle bracket
Either the left angle bracket (<) or the right angle bracket (>). In the portable character set, these characters are referred to by the names <less-than-sign> and <greater-than-sign>.

anonymous union
An unnamed object whose type is an unnamed union.

ANSI
See American National Standards Institute.

AP
See application program.

API
See application programming interface.

application
One or more computer programs or software components that provide a function in direct support of a specific business process or processes.

application generator
An application development tool that creates applications, application components (panels, data, databases, logic, interfaces to system services), or complete application systems from design specifications.

application program (AP)
A complete, self-contained program, such as a text editor or a web browser, that performs a specific task for the user, in contrast to system software, such as the operating system kernel, server processes, and program libraries.

application programming interface (API)
An interface that allows an application program that is written in a high-level language to use specific data or functions of the operating system or another program.

archive library
A facility for grouping application-program object files. The archive library file, when created for application-program object files, has a special symbol table for members that are object files.

argument
A value passed to or returned from a function or procedure at run time.

argument declaration
See also parameter declaration.

arithmetic object
An integral object or objects having the float, double, or long double type.

array
In programming languages, an aggregate that consists of data objects, with identical attributes, each of which can be uniquely referenced by subscripting. See also scalar.

array element
One of the data items in an array.

ASCII
See American Standard Code for Information Interchange.

assembler
A computer program that converts assembly language instructions into object code.

Assembler H
An IBM licensed program that translates symbolic assembler language into binary machine language.

assembler user exit
A routine to tailor the characteristics of an enclave prior to its establishment.

assembly language
A symbolic programming language that represents machine instructions of a specific architecture.

assignment expression
An expression that assigns the value of the right operand expression to the left operand variable and has as its value the value of the right operand.

automatic call library
A group of modules that are used as secondary input to the binder to resolve external symbols left undefined after all the primary input has been processed. The automatic call library can contain: object modules, with or without binder control statements; load modules; and runtime routines.

automatic library call
The process by which the binder resolves external references by including additional members from the automatic call library.

automatic storage
Storage that is allocated on entry to a routine or block and is freed when control is returned. See also dynamic storage.

auto storage class specifier
A specifier that enables the programmer to define a variable with automatic storage; its scope is restricted to the current block.

B

background process
A process that does not require operator intervention but can be run by the computer while the workstation is used to do other work. See also foreground process.

background processing
A mode of program execution in which the shell does not wait for program completion before prompting the user for another command.

backslash
The character \. The backslash enables a user to escape the special meaning of a character. That is, typing a backslash before a character tells the system to ignore any special meaning the character might have.

binary expression
An expression containing two operands and one operator.

binary stream
A sequence of characters that corresponds on a one-to-one basis with the characters in the file. No character translation is performed on binary streams.

binder

1. The z/OS program that processes the output of language translators and compilers into an executable program (a load module or program object). The binder replaces the linkage editor and batch loader. See also prelinker.
2. See linkage editor.

bit field
A member of a structure or union that contains 1 or more named bits.

bitwise operator
An operator that manipulates the value of an object at the bit level.

blank character

1. One of the characters that belong to the blank character class as defined via the LC_CTYPE category in the current locale. In the POSIX locale, a blank character is either a tab or a space character.
2. A graphic representation of the space character.

block

1. A string of data elements recorded, processed, or transmitted as a unit. The elements can be characters, words, or physical records.
2. In programming languages, a compound statement that coincides with the scope of at least one of the declarations contained within it. A block may also specify storage allocation or segment programs for other purposes.

block statement
In the C language, a group of data definitions, declarations, and statements that are located between a left brace and a right brace that are processed as a unit. The block statement is considered to be a single, C-language statement.

boundary alignment
The position in main storage of a fixed-length field, such as halfword or doubleword, which is aligned on an integral boundary for that unit of information. For example, a word boundary alignment stores the object in a storage address evenly divisible by four.

brace
Either of the characters left brace ({) and right brace (}). When an object is enclosed in braces, the left brace immediately precedes the object and the right brace immediately follows it.

bracket
Either of the characters left bracket ([) and right bracket (]).

break statement
A C control statement that contains the keyword break and a semicolon (;). It is used to end an iterative or a switch statement by exiting from it at any point other than the logical end. Control is passed to the first statement after the iteration or switch statement.

built-in
In programming languages, pertaining to a language object that is defined in the programming language specification.

built-in function
A function that is predefined by the compiler and whose code is incorporated directly into the compiled object rather than called at run time. See also function.

byte-oriented stream
A byte-oriented stream refers to a stream which only single byte input/output is allowed.

C

callable service
A program service provided through a programming interface.

call chain
A trace of all active routines and subroutines, such as the names of routines and the locations of save areas, that can be constructed from information included in a system dump.

caller
A function that calls another function.

cancelability point
A specific point within the current thread that is enabled to solicit cancel requests.

carriage return character
A character that in the output stream indicates that printing should start at the beginning of the same physical line in which the carriage-return character occurred.

case clause
In a C switch statement, a CASE label followed by any number of statements.

case label
The word case followed by a constant expression and a colon. When the selector is evaluated to the value of the constant expression, the statements following the case label are processed.

cast expression
An expression that converts or reinterprets its operand.

cast operator
An operator that is used for explicit type conversions.

cataloged procedure
A set of job control language (JCL) statements that has been placed in a library and that is retrievable by name.

CCS
See coded character set.

character

1. A sequence of one or more bytes representing a single graphic symbol or control code.
2. In a computer system, a member of a set of elements that is used for the representation, organization, or control of data.

character class
A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the LC_CTYPE category in the current locale.

character constant
The actual character value (a symbol, quantity, or constant) in a source program that is itself data, instead of a reference to a field that contains the data.

character set
A defined set of characters with no coded representation assumed that can be recognized by a configured hardware or software system. A character set can be defined by alphabet, language, script, or any combination of these items.

character special file
An interface file that provides access to an input or output device, which uses character I/0 instead of block I/0.

character string
A contiguous sequence of characters terminated by and including the first null byte.

child
A node that is subordinate to another node in a tree structure. Only the root node is not a child.

child enclave
The nested enclave created as a result of certain commands being issued from a parent enclave. See also nested enclave, parent enclave.

child process
A process that is created by a parent process and that shares the resources of the parent process to carry out a request.

C language
A language used to develop application programs in compact, efficient code that can be run on different types of computers with minimal change.

C library
A system library that contains common C language subroutines for file access, string operations, character operations, memory allocation, and other functions.

CLIST
See command list.

COBOL
See Common Business Oriented Language.

coded character set (CCS)
A set of unambiguous rules that establishes a character set and the one-to-one relationships between the characters of the set and their coded representations.

code element set
The result of applying rules that map a numeric code value to each element of a character set. An element of a character set may be related to more than one numeric code value but the reverse is not true. However, for state-dependent encodings the relationship between numeric code values to elements of a character set may be further controlled by state information. The character set may contain fewer elements than the total number of possible numeric code values; that is, some code values may be unassigned. X/Open.

code generator
The part of the compiler that physically generates the object code.

code page
A particular assignment of code points to graphic characters. Within a given code page, a code point can have only one specific meaning. A code page also identifies how undefined code points are handled. See also code point.

code point

1. An identifier in an alert description that represents a short unit of text. The code point is replaced with the text by an alert display program.
2. A unique bit pattern that represents a character in a code page. See also code page.

collating element
The smallest entity used to determine the logical ordering of strings. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the LC_COLLATE category in the current locale determines the current set of collating elements. See also collating sequence.

collating sequence
The relative ordering of collating elements as determined by the setting of the LC_COLLATE category in the current locale. The character order, as defined for the LC_COLLATE category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order.

collation
The logical ordering of characters and strings according to defined rules.

collection
An abstract class without any ordering, element properties, or key properties.

Collection Class Library
A complete set of abstract data structure such as trees, stacks, queues, and linked lists.

column position
A unit of horizontal measure related to characters in a line. It is assumed that each character in a character set has an intrinsic column width independent of any output device. Each printable

character in the portable character set has a column width of one. The standard utilities, when used as described in this document set, assume that all characters have integral column widths. The column width of a character is not necessarily related to the internal representation of the character (numbers of bits or bytes). The column position of a character in a line is defined as one plus the sum of the column widths of the preceding characters in the line. Column positions are numbered starting from 1. X/Open.

comma expression
An expression that contains two operands separated by a comma. Although the compiler evaluates both operands, the value of the right operand is the value of the expression. If the left operand produces a value, the compiler discards this value.

command
A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

command list (CLIST)
A language for performing TSO tasks.

Common Business Oriented Language (COBOL)
A high-level programming language that is used primarily for commercial data processing.

compilation unit
A portion of a computer program sufficiently complete to be compiled correctly.

compiler option
A keyword that can be specified to control certain aspects of compilation. Compiler options can control the nature of the load module generated by the compiler, the types of printed output to be produced, the efficient use of the compiler, and the destination of error messages.

condition
An expression that can be evaluated as true, false, or unknown. It can be expressed in natural language text, in mathematically formal notation, or in a machine-readable language.

conditional expression
A compound expression that contains a condition (the first expression), an expression to be evaluated if the condition has a nonzero value (the second expression), and an expression to be evaluated if the condition has the value zero (the third expression).

condition handler
A user-written routine or language-specific routine (such as a PL/ION-unit or C signal() function call) invoked by the Language Environment® condition manager to respond to conditions.

condition manager
The condition manager is the part of the common execution environment that manages conditions by invoking various user-written and language-specific condition handlers.

constant
A language element that specifies an unchanging value. Constants are classified as string constants or numeric constants.

constant expression
An expression that has a value that can be determined during compilation and that does not change during the running of the program.

constant propagation
An optimization technique where constants used in an expression are combined and new ones are generated. Mode conversions are done to allow some intrinsic functions to be evaluated at compile time.

constructed reentrancy
The attribute of applications that contain external data and require additional processing to make them reentrant. See also natural reentrancy.

control character
A character whose occurrence in a particular context initiates, modifies, or stops a control function.

controlling process
A session leader that has control of a terminal.

controlling terminal
The active workstation from which the process group for that process was started. Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session.

control section (CSECT)
The part of a program specified by the programmer to be a relocatable unit, all elements of which are to be loaded into adjoining main storage locations.

control statement
In programming languages, a statement that is used to interrupt the continuous sequential processing of programming statements. Conditional statements such as IF, PAUSE, and STOP are examples of control statements.

conversion

1. In programming languages, the transformation between values that represent the same data item but belong to different data types. Information may be lost because of conversion since accuracy of data representation varies among different data types.
2. The process of changing from one form of representation to another. Changing a code point that is assigned to a character in one code page to its corresponding code point in another code page is an example of conversion.

Coordinated Universal Time (UTC)
The international standard of time that is kept by atomic clocks around the world.

cross-compiler
A compiler that produces executable files that run on a platform other than the one on which the compiler is installed.

CSECT
See control section.

current working directory
See working directory.

cursor
A reference to an element at a specific position in a data structure.

D

data abstraction
A data type with a private representation and a public set of operations (functions or operators) which restrict access to that data type to that set of operations.

data definition (DD)
A program statement that describes the features of, specifies relationships of, or establishes the context of data. A data definition reserves storage and can provide an initial value.

data definition name (ddname)
The name of a data definition (DD) statement that corresponds to a data control block that contains the same name.

data definition statement (DD statement)
A job control statement that is used to define a data set for use by a batch job step, started task or job, or an online user.

data member
The smallest possible piece of complete data. Elements are composed of data members.

data object
An element of data structure such as a file, an array, or an operand that is needed for the execution of an application.

data set
The major unit of data storage and retrieval, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access.

data stream
The commands, control codes, data, or structured fields that are transmitted between an application program and a device such as printer or nonprogrammable display station.

data structure
In Open Source Initiative (OSI), the syntactic structure of symbolic expressions and their storage allocation characteristics.

data type
A category that identifies the mathematical qualities and internal representation of data and functions.

Data Window Services (DWS)
Services provided as part of the Callable Services Library that allow manipulation of data objects such as VSAM linear data sets and temporary data objects known as TEMPSPACE.

DBCS
See double-byte character set.

DD
See data definition.

ddname
See data definition name.

DD statement
See data definition statement.

dead code
Code that is never referenced, or that is always branched over.

dead store
A store into a memory location that will later be overwritten by another store without any intervening loads. In this case, the earlier store can be deleted.

decimal constant
A numerical data type used in standard arithmetic operations. Decimal constants can contain any digits 0 through 9. See also integer constant.

decimal overflow
A condition that occurs when one or more nonzero digits are lost because the destination field in a decimal operation is too short to contain the results.

declaration

1. In the C language, a description that makes an external object or function available to a function or a block statement.
2. A statement that establishes the names and characteristics of data objects and functions used in a program.

default clause
In the C languages, within a switch statement, the keyword default followed by a colon, and one or more statements. When the conditions of the specified case labels in the switch statement do not hold, the default clause is chosen.

default initialization
The initial value assigned to a data object by the compiler if no initial value is specified by the programmer. In C language, external and static variables receive a default initialization of zero, while the default initialization for auto and register variables is undefined.

definition
A declaration that reserves storage and can provide an initial value for a data object or define a function.

degree
The number of children of a node.

dereference
In the C language, to apply the unary operator * to a pointer to access the object the pointer points to. See also indirection.

descriptor
A PL/I control block that holds information such as string lengths, array subscript bounds, and area sizes, and is passed from one PL/I routine to another during run time.

device
A piece of equipment such as a workstation, printer, disk drive, tape unit, or remote system.

difference
Given two sets A and B, the set of all elements contained in A but not in B (A-B).

digraph
A combination of two keystrokes used to represent unavailable characters in a C source program. Digraphs are read as tokens during the preprocessor phase.

directive
A control statement that directs the operation of a feature and is recognized by a preprocessor or other tool. See also pragma.

directory

1. The part of a partitioned data set that describes the members in the data set.
2. In a hierarchical file system, a grouping of related files.

display
To direct the output to the user's terminal. If the output is not directed to the terminal, the results are undefined.

do statement
A looping statement that contains the keyword do, followed by a statement (the action), the keyword while, and an expression in parentheses (the condition).

dot
A symbol (.) that indicates the current directory in a relative path name. See also period.

double-byte character set (DBCS)
A set of characters in which each character is represented by 2 bytes. These character sets are commonly used by national languages, such as Japanese and Chinese, that have more symbols than can be represented by a single byte. See also single-byte character set.

double-precision
Pertaining to the use of two computer words to represent a number in accordance with the required precision.

doubleword
A contiguous sequence of bits or characters that comprises two computer words and is capable of being addressed as a unit. See also halfword, word.

DSA
See dynamic storage area.

DWS
See Data Window Services.

dynamic
Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time.

dynamic access
A process where records can be accessed sequentially or randomly, depending on the form of the input/output request. See also access mode.

dynamic allocation
Assignment of system resources to a program when the program is executed rather than when it is loaded into main storage.

dynamic binding
The act of resolving references to external variables and functions at run time.

dynamic storage
An area of storage that is explicitly allocated by a program or procedure while it is running. See also automatic storage.

dynamic storage area (DSA)
A type of storage allocation in which storage is assigned to a program or application at run time.

E

EBCDIC
See Extended Binary Coded Decimal Interchange Code.

effective group ID
An attribute of a process that is used in determining various permissions, including file access permissions. This value is subject to change during the process lifetime.

element
The smallest unit in a table, array, list, set, or other structure. Examples of an element are a value in a list of values and a data field in an array.

element equality
A relation that determines if two elements are equal.

element occurrence
A single instance of an element in a collection. In a unique collection, element occurrence is synonymous with element value.

element value
All the instances of an element with a particular value in a collection. In a non-unique collection, an element value may have more than one occurrence. In a unique collection, element value is synonymous with element occurrence.

else clause
The part of an if statement that contains the keyword 'else' followed by a statement. The else clause provides an action that is started when the if condition evaluates to a value of 0 (false).

empty line
A line consisting of only a newline character. X/Open.

empty string
A character array whose first element is a null character.

encapsulation
In object-oriented programming, the technique that is used to hide the inherent details of an object, function, or class from client programs.

entry point
The address or label of the first instruction processed or entered in a program, routine, or subroutine.There might be a number of different entry points, each corresponding to a different function or purpose.

enum constant
See enumeration constant.

enumeration constant (enum constant)
In the C language, an identifier, with an associated integer value, defined in an enumerator. An enumeration constant may be used anywhere an integer constant is allowed.

enumeration data type
A data type that represents a set of values that a user defines.

enumeration tag
The identifier that names an enumeration data type.

enumeration type
A data type that defines a set of enumeration constants.

enumerator
An enumeration constant and its associated value.

equivalence class
A grouping of characters or character strings that are considered equal for purposes of collation. For example, many languages place an uppercase character in the same equivalence class as its lowercase form, but some languages distinguish between accented and unaccented character forms for the purpose of collation.

escape sequence
A string of bit combinations that is used to escape from normal data, such as text code points, into control information.

exception
A condition or event that cannot be handled by a normal process.

executable file
A file that contains programs or commands that perform operations on actions to be taken.

executable program
A program in a form suitable for execution by a computer. The program can be an application or a shell script.

Extended Binary Coded Decimal Interchange Code (EBCDIC)
A coded character set of 256 8-bit characters developed for the representation of textual data. See also American Standard Code for Information Interchange.

extended-precision
Pertains to the use of more than two computer words to represent a floating point number in accordance with the required precision. For example, in z/OS, four computer words are used for an extended-precision number.

extension
An element or function not included in the standard language.

F

FIFO special file
A type of file with the property that data written to such a file is read on a first-in-first-out (FIFO) basis.

file descriptor
A positive integer or a data structure that uniquely identifies an open file for the purpose of file access.

file mode
An object containing the file permission bits and other characteristics of a file.

file permission bit
Information about a file that is used, along with other information, to determine whether a process has read, write, or execute permission to a file. The use of file permission bits is described in file access permissions.

file scope
A property of a file name that is declared outside all blocks, classes, and function declarations and that can be used after the point of declaration in a source file.

filter
A command that reads standard input data, modifies the data, and sends it to standard output. A pipeline usually has several filters.

flat collection
A collection that has no hierarchical structure.

float constant

1. A constant representing a non-integral number.
2. A number containing a decimal point, an exponent, or both a decimal point and an exponent. The exponent contains an "e" or "E," an optional sign (+ or -), and one or more digits (0 through 9).

footprint
The amount of computer storage that is occupied by a computer program. For example, if a program occupies a large amount of storage, it has a large footprint.

foreground process
A process that must be completed before another command is issued. See also background process.

foreground process group
A group whose member processes have privileges that are denied to background processes when the controlling terminal is being accessed. Each controlling terminal can have only one foreground process group.

form-feed character
A character in the output stream that indicates that printing should start on the next page of an output device. The form-feed character is designated by '\f' in the C language. If the form-feed character is not the first character of an output line, the result is unspecified. X/Open.

for statement
A looping statement that contains the word for followed by a list of expressions enclosed in parentheses (the condition) and a statement (the action). Each expression in the parenthesized list is separated by a semicolon, which cannot be omitted.

forward declaration
A declaration of a class or function made earlier in a compilation unit, so that the declared class or function can be used before it has been defined.

freestanding application
An application that is created to run without the run-time environment or library with which it was developed.

free store
Dynamically allocated memory. New and delete are used to allocate and deallocate free store.

function
A named group of statements that can be called and evaluated and can return a value to the calling statement. See also built-in function.

function call
An expression that transfers the path of execution from the current function to a specified function (the called function). A function call contains the name of the function to which control is transferred and a parenthesized list of values.

function declarator
The part of a function definition that names the function, provides additional information about the return value of the function, and lists the function parameters.

function definition
The complete description of a function. A function definition contains an optional storage class specifier, an optional type specifier, a function declarator, optional parameter declarations, and a block statement (the function body).

function prototype
A function declaration that provides type information for each parameter. It is the first line of the function (header) followed by a semicolon (;). The declaration is required by the compiler at the time that the function is declared, so that the compiler can check the type.

function scope
Labels that are declared in a function have function scope and can be used anywhere in that function after their declaration.

G

GCC
See GNU Compiler Collection.

GDDM
See Graphical Data Display Manager.

Generalized Object File Format (GOFF)
This object module format extends the capabilities of object modules so that they can contain more information.

global
Pertaining to information available to more than one program or subroutine. See also local.

global variable
A symbol defined in one program module that is used in other program modules that are independently compiled.

GMT
See Greenwich mean time.

GNU Compiler Collection (GCC)
An open source collection of compilers supporting C, C++, Objective-C, Ada, Java™, and Fortran.

GOFF
See Generalized Object File Format.

Graphical Data Display Manager (GDDM)
An IBM computer-graphics system that defines and displays text and graphics for output on a display or printer.

graphic character
A visual representation of a character, other than a control character, that is normally produced by writing, printing, or displaying.

Greenwich mean time (GMT)
The mean solar time at the meridian of Greenwich, England.

H

halfword
A contiguous sequence of bits or characters that constitutes half a computer word and can be addressed as a unit. See also doubleword, word.

hash function
A function that determines which category, or bucket, to put an element in. A hash function is needed when implementing a hash table.

hash table

1. A data structure that divides all elements into (preferably) equal-sized categories, or buckets, to allow quick access to the elements. The hash function determines which bucket an element belongs in.
2. A table of information that is accessed by way of a shortened search key (the hash value). The use of a hash table minimizes average search time.

header file
See include file.

heap storage
An area of storage used for allocation of storage that has a lifetime that is not related to the execution of the current routine. The heap consists of the initial heap segment and zero or more increments.

hexadecimal constant
A constant, usually starting with special characters, that contains only hexadecimal digits.

High Level Assembler
An IBM licensed program that translates symbolic assembler language into binary machine language.

hiperspace memory file
A type of file that is stored in a single buffer in an address space, with the rest of the data being kept in a hiperspace. In contrast, for regular files, all the file data is stored in a single address space.

hook
A location in a compiled program where the compiler has inserted an instruction that allows programmers to interrupt the program (by setting breakpoints) for debugging purposes.

hybrid code
Program statements that have not been internationalized with respect to code page, especially where data constants contain variant characters. Such statements can be found in applications written in older implementations of MVS™, which required syntax statements to be written using code page IBM-1047 exclusively. Such applications cannot be converted from one code page to another using iconv().

I

ID
See identifier.

identifier (ID)
One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element.

if statement
A conditional statement that specifies a condition to be tested and the action to be taken if the condition is satisfied.

ILC

1. See interlanguage communication.
2. See interlanguage call.

implementation-defined
Pertaining to behavior that is defined by the compiler rather than by a language standard. Programs that rely on implementation-defined behavior may behave differently when compiled with different compilers. See also undefined behavior.

IMS
See Information Management System.

include directive
A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

include file
A text file that contains declarations that are used by a group of functions, programs, or users.

incomplete type
A type that has no value or meaning when it is first declared. There are three incomplete types: void, arrays of unknown size and structures, and unions of unspecified content.

indirection

1. A mechanism for connecting objects by storing, in one object, a reference to another object. See also dereference.
2. In the C language, the application of the unary operator * to a pointer to access the object to which the pointer points.

induction variable
A controlling variable of a loop.

Information Management System (IMS)
Any of several system environments that have a database manager and transaction processing that can manage complex databases and terminal networks.

initial heap
A heap that is controlled by the HEAP run-time option and designated by a heap_id of 0.

initializer
An expression used to initialize data objects.

inline
To replace a function call with a copy of the function's code during compilation.

inline function
A function whose actual code replaces a function call. A function that is both declared and defined in a class definition is an example of an inline function. Another example is one which you explicitly declared inline by using the keyword inline. Both member and non-member functions can be inlined.

input stream
A sequence of control statements and data submitted to an operating system by an input device.

instruction
A program statement that specifies an operation to be performed by the computer, along with the values or locations of operands. This statement represents the programmer's request to the processor to perform a specific operation.

instruction scheduling
An optimization technique that reorders instructions in code to minimize execution time.

integer constant
A decimal, octal, or hexadecimal constant. See also decimal constant.

integral object
A character object, an object having an enumeration type, an object having variations of the type int, or an object that is a bit field.

Interactive System Productivity Facility (ISPF)
An IBM licensed program that serves as a full-screen editor and dialog manager. Used for writing application programs, it provides a means of generating standard screen panels and interactive dialogs between the application programmer and the terminal user.

interlanguage call (ILC)
A call to a procedure or function made by a program written in one language to a procedure or function coded in a different language.

interlanguage communication (ILC)
The ability of routines written in different programming languages to communicate. ILC support enables the application writer to readily build applications from component routines written in a variety of languages.

interoperability
The ability of a computer or program to work with other computers or programs.

interprocess communication (IPC)
The process by which programs send messages to each other. Sockets, semaphores, signals, and internal message queues are common methods of interprocess communication.

IPC
See interprocess communication.

ISPF
See Interactive System Productivity Facility.

iteration
The repetition of a set of computer instructions until a condition is satisfied.

J

JCL
See job control language.

job control language (JCL)
A command language that identifies a job to an operating system and describes the job requirements.

K

kernel
The part of an operating system that contains programs for such tasks as input/output, management and control of hardware, and the scheduling of user tasks.

keyword

1. One of the predefined words of a programming language, artificial language, application, or command. See also operand, parameter.
2. A symbol that identifies a parameter in job control language (JCL).

L

label
An identifier within or attached to a set of data elements.

last element
The element visited last in an iteration over a collection. Each collection has its own definition for last element. For example, the last element of a sorted set is the element with the largest value.

leaf
In a tree, an entry or node that has no children.

library

1. A collection of model elements, including business items, processes, tasks, resources, and organizations.
2. A set of object modules that can be specified in a link command.

linkage
Refers to the binding between a reference and a definition. A function has internal linkage if the function is defined inline as part of the class, is declared with the inline keyword, or is a non-member function declared with the static keyword. All other functions have external linkage.

linkage editor
A computer program for creating load modules from one or more object modules or load modules by resolving cross-references among the modules and, if necessary, adjusting addresses.

linker
A program that resolves cross-references among separately compiled object modules and then assigns final addresses to create a single executable program.

link pack area (LPA)
The portion of virtual storage below 16 MB that contains frequently used modules.

literal
A symbol or a quantity in a source program that is itself data, rather than a reference to data.

loader
A program that copies an executable file into main storage so that the file can be run.

load module
A program in a form suitable for loading into main storage for execution.

local

1. Pertaining to information that is defined and used only in one subdivision of a computer program. See also global.
2. In programming languages, pertaining to the relationship between a language object and a block such that the language object has a scope contained in that block.

local custom
A convention of a geographical area or territory for such things as date, time, and currency formats. X/Open.

locale
A setting that identifies language or geography and determines formatting conventions such as collation, case conversion, character classification, the language of messages, date and time representation, and numeric representation.

local scope
A name declared in a block that has local scope and can only be used in that block.

loop unrolling
An optimization that increases the step of a loop, and duplicates the expressions within a loop to reflect the increase in the step. This can improve instruction scheduling and memory access time.

LPA
See link pack area.

lvalue
An expression that represents a data object that can be viewed, tested, and changed. An lvalue is usually the left operand in an assignment expression.

M

macro
An instruction that causes the execution of a predefined sequence of instructions.

macro call
See macro.

main function
A function that has the identifier main. Each program must have exactly one function named main. The main function is the first user function that receives control when a program starts to run.

makefile
In UNIX, a text file containing a list of an application's parts. The make utility uses makefiles to maintain application parts and dependencies.

make utility
A utility that maintains all of the parts and dependencies for an application. The make utility uses a makefile to keep the parts of a program synchronized. If one part of an application changes, the make utility updates all other files that depend on the changed part.

manipulator
A value that can be inserted into streams or extracted from streams to affect or query the behavior of the stream.

method file

1. For ASCII locales, a file that defines the method functions to be used by C runtime locale-sensitive interfaces. A method file also identifies where the method functions can be found. IBM supplies several method files used to create its standard set of ASCII locales. Other method files can be created to support customized or user-created code pages. Such customized method files replace IBM-supplied charmap method functions with user-written functions.
2. A file that allows users to indicate to the localedef utility where to look for user-provided methods for processing user-designed code pages.

migrate
To install a new version or release of a program to replace an earlier version or release.

module
A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading.

multibyte character
A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

multibyte control
See escape sequence.

multicharacter collating element
A sequence of two or more characters that collate as an entity. For example, in some coded character sets, an accented character is represented by a non-spacing accent, followed by the letter. Other examples are the Spanish elements ch and ll. X/Open.

multiprocessor
A processor complex that has more than one central processor.

multitasking
A mode of operation in which two or more tasks can be performed at the same time.

mutex
See mutual exclusion.

mutex attribute object
A type of attribute object with which a user can manage mutual exclusion (mutex) characteristics by defining a set of variables to be used during its creation. A mutex attribute object eliminates the need to redefine the same set of characteristics for each mutex object created. See also mutual exclusion.

mutex object
An identifier for a mutual exclusion (mutex).

mutual exclusion (mutex)
A flag used by a semaphore to protect shared resources. The mutex is locked and unlocked by threads in a program. See also mutex attribute object.

N

namespace
A category used to group similar types of identifiers.

natural reentrancy
The attribute of applications that contain no static external data and do not require additional processing to make them reentrant. See also constructed reentrancy.

nested enclave
A new enclave created by an existing enclave. The nested enclave that is created must be a new main routine within the process. See also child enclave, parent enclave.

newline character (NL)
A control character that causes the print or display position to move down one line.

nickname
See alias.

NL
See newline character.

nonprinting character
See control character.

NUL
See null character.

null character (NUL)
A control character with the value of X'00' that represents the absence of a displayed or printed character.

null pointer
The value that is obtained by converting the number 0 into a pointer; for example, (void *) 0. The C language guarantees that this value will not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error.

null statement
A statement that consists of a semicolon.

null string
A character or bit string with a length of zero.

null value
A parameter position for which no value is specified.

null wide-character code
A wide-character code with all bits set to zero.

number sign
The character #, which is also referred to as the hash sign.

O

object

1. A region of storage. An object is created when a variable is defined. An object is destroyed when it goes out of scope.
2. In object-oriented design or programming, a concrete realization (instance) of a class that consists of data and the operations associated with that data. An object contains the instance data that is defined by the class, but the class owns the operations that are associated with the data.

object module
A set of instructions in machine language that is produced by a compiler or assembler from a subroutine or source module and can be input to the linking program. The object module consists of object code.

octal constant
The digit 0 (zero) followed by any digits 0 through 7.

open file
A file that is currently associated with a file descriptor.

operand
An entity on which an operation is performed.

operating system (OS)
A collection of system programs that control the overall operation of a computer system.

operator precedence
In programming languages, an order relationship that defines the sequence of the application of operators with an expression.

orientation
The orientation of a stream refers to the type of data which may pass through the stream. A stream without orientation is one on which no stream I/O has been performed.

OS
See operating system.

overflow
The condition that occurs when data cannot fit in the designated field.

overlay
The technique of repeatedly using the same areas of internal storage during different stages of a program. Unions are used to accomplish this in C.

P

parameter (parm)
A value or reference passed to a function, command, or program that serves as input or controls actions. The value is supplied by a user or by another program or process. See also keyword, operand.

parameter declaration
The description of a value that a function receives. A parameter declaration determines the storage class and the data type of the value. See also argument declaration.

parent enclave
The enclave that issues a call to system services or language constructs to create a nested (or child) enclave. See also child enclave, nested enclave.

parent process
A process that is created to carry out a request or set of requests. The parent process, in turn, can create child processes to process requests for the parent.

parent process ID (PPID)
An attribute of a new process identifying the parent of the process. The parent process ID of a process is the process ID of its creator for the lifetime of the creator. After the creator's lifetime has ended, the parent process ID is the process ID of an implementation-dependent system process.

parm
See parameter.

partitioned concatenation
The allocation of partitioned data sets (PDSs), partitioned data sets extended (PDSEs), UNIX file directories, or any combination of these such that the basic partitioned access method (BPAM) retrieves them as a single data set.

partitioned data set (PDS)
A data set on direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data. See also sequential data set.

partitioned data set extended (PDSE)
A data set that contains an indexed directory and members that are similar to the directory and members of partitioned data sets (PDSs). See also library.

path name
A name that specifies all directories leading to a file plus the file name itself.

path name resolution
The process of resolving a path name to a particular file in a file hierarchy. There may be multiple path names that resolve to the same file. X/Open.

pattern
A sequence of characters used either with regular expression notation or for path name expansion, as a means of selecting various characters strings or path names, respectively. The syntaxes of the two patterns are similar, but not identical.

PDS
See partitioned data set.

PDSE
See partitioned data set extended.

period
The symbol ".". The term dot is used for the same symbol when referring to a web address or file extension. This character is named <period> in the portable character set. See also dot.

permission
The ability to access a protected object, such as a file or directory. The number and meaning of permissions for an object are defined by the access control list.

persistent environment
An environment that once created by the user may be used repeatedly without incurring the overhead of initialization and termination for each call. The environment remains available until explicitly terminated by the user.

PGID
See process group ID.

PID
See process ID.

platform
The combination of an operating system and hardware that makes up the operating environment in which a program runs.

pointer
A data element or variable that holds the address of a data object or a function. See also scalar.

portability

1. The ability of a program to run on more than one type of computer system without modification.
2. The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

portable character set
A set of characters, specified in POSIX 1003.2, section 4, that must be supported by conforming implementations.

portable file name character set
The set of characters from which portable file names must be constructed to be portable across implementations conforming to the ISO POSIX-1 standard and to ISO/IEC 9945.

positional parameter
A parameter that must appear in a specified location, relative to other parameters.

PPID
See parent process ID.

pragma
A standardized form of comment which has meaning to a compiler. A pragma usually conveys non-essential information, often intended to help the compiler to optimize the program. See also directive.

precedence
The priority system for grouping different types of operators with their operands.

predefined macro
An identifier predefined by the compiler, which will be expanded by the preprocessor during compilation.

preinitialization
A process by which an environment or library is initialized once and can then be used repeatedly to avoid the inefficiency of initializing the environment or library each time it is needed.

prelinker
A utility that preprocesses an object for certain programs. See also binder.

preprocessor
A routine that performs initial processing and translation of source code or data prior to compiling the source code or processing the data in another program such as an emulator.

preprocessor directive
In the C language, a statement that begins with the symbol # and is interpreted by the preprocessor during compilation.

preprocessor statement
In the C language, a statement that begins with the symbol # and contains instructions that the preprocessor can interpret.

primary expression

1. Literals, names, and names qualified by the :: (scope resolution) operator.
2. Any of the following types of expressions: a) identifiers, b) parenthesized expressions, c) function calls, d) array element specifications, e) structure member specifications, or f) union member specifications.

process

1. An address space and single thread of control that executes within that address space, and its required system resources. A process is created by another process issuing the fork() function. The process that issues the fork() function is known as the parent process, and the new process created by the fork() function is known as the child process.
2. An instance of a program running on a system and the resources that it uses.

process group
A collection of processes in a system that is identified by a process group ID.

process group ID (PGID)
The unique identifier representing a process group during its lifetime. A process group ID is a positive integer that is not reused by the system until the process group lifetime ends.

process group lifetime
A period of time that begins when a process group is created and ends when the last remaining process in the group leaves the group because either it is the end of the last process' lifetime or the last remaining process is calling the setsid() or setpgid() functions. X/Open. ISO.1.

process ID (PID)
The unique identifier that represents a process. A process ID is a positive integer and is not reused until the process lifetime ends.

process lifetime
The period of time that begins when a process is created and ends when the process ID is returned to the system. X/Open. ISO.1. After a process is created with a fork() function, it is considered active. Its thread of control and address space exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources, such as the process ID, are still in use. When another process executes a wait() or waitpid() function for an inactive process, the remaining resources are returned to the system. The last resource to be returned to the system is the process ID. At this time, the lifetime of the process ends.

profiling
A performance analysis process that is based on statistics for the resources that are used by a program or application.

program object
All or part of a computer program in a form suitable for loading into virtual storage for execution. Program objects are stored in partitioned data set extended (PDSE) program libraries and have fewer restrictions than load modules. Program objects are produced by the binder.

program unit
See compilation unit.

prototype
A function declaration or definition that includes both the return type of the function and the types of its parameters.

Q

QMF
See Query Management Facility.

qualified name

1. A data set name consisting of a string of names separated by periods; for example, TREE.FRUIT.APPLE is a qualified name.

qualified type name
A name used to reduce complex class name syntax by using typedefs to represent qualified class names.

Query Management Facility (QMF)
An IBM query and report writing facility that supports a variety of tasks such as data entry, query building, administration, and report analysis.

queue
A data structure for processing work in which the first element added to the queue is the first element processed. This order is referred to as first-in first-out (FIFO).

quotation mark
The characters " and '.

R

radix character
The character that separates the integer part of a number from the fractional part. X/Open .

random access
An access mode in which records can be referred to, read from, written to, or removed from a file in any order.

real group ID
The attribute of a process that, at the time of process creation, identifies the group of the user who created the process. This value is subject to change during the process lifetime.

real user ID
The attribute of a process that, at the time a process is created, identifies the user who created the process.

reason code
A value used to indicate the specific reason for an event or condition.

reassociation
An optimization technique that rearranges the sequence of calculations in a subscript expression producing more candidates for common expression elimination.

redirection
In a shell, a method of associating files with the input or output of commands.

reentrant
The attribute of a program or routine that allows the same copy of the program or routine to be used concurrently by two or more tasks.

refresh
To ensure that the information on the user's terminal screen is up-to-date.

register variable
A variable defined with the register storage class specifier. Register variables have automatic storage.

regular expression

1. A set of characters, meta characters, and operators that define a string or group of strings in a search pattern.
2. A string containing wildcard characters and operations that define a set of one or more possible strings.
3. A mechanism for selecting specific strings from a set of character strings.

regular file
A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. [POSIX.1]

relation
An unordered flat collection class that uses keys, allows for duplicate elements, and has element equality.

relative path name
A string of characters that is used to refer to an object and that starts at some point in the directory hierarchy other than the root. The starting point is frequently a user's current directory.

reserved word
A word that is defined by a programming language and that cannot be used as an identifier or changed by the user.

residency mode (RMODE)
In z/OS, a program attribute that refers to where a module is prepared to run. RMODE can be 24 or ANY. ANY refers to the fact that the module can be loaded either above or below the 16M line. RMODE 24 means the module expects to be loaded below the 16M line.

reverse solidus

RMODE
See residency mode.

runtime environment
A set of resources that are used to run a program or process.

runtime library
A compiled collection of functions whose members can be referred to by an application program at run time.

S

SBCS
See single-byte character set.

scalar
An arithmetic object, an enumerated object, or a pointer to an object.

scope
A part of a source program in which an object is defined and recognized.

SDK
See software development kit.

semaphore
An object used by multi-threaded applications for signaling purposes and for controlling access to serially reusable resources. Processes can be locked to a resource with semaphores if the processes follow certain programming conventions.

sequence
A sequentially ordered flat collection.

sequential access
The process of referring to records one after another in the order in which they appear on the file. See also access mode.

sequential concatenation
The allocation of sequential data sets, partitioned data set (PDS) members, partitioned data set extended (PDSE) members, UNIX files, or any combination of these such that the system retrieves them as a single, sequential, data set.

sequential data set
A data set whose records are organized based on their successive physical positions, such as on magnetic tape. See also partitioned data set.

session
A collection of process groups established for job control purposes.

shell
A software interface between users and an operating system. Shells generally fall into one of two categories: a command line shell, which provides a command line interface to the operating system; and a graphical shell, which provides a graphical user interface (GUI).

signal

1. A mechanism by which a process can be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes.
2. In operating system operations, a method of inter-process communication that simulates software interrupts.
3. A condition that might or might not be reported during program execution. For example, a signal can represent erroneous arithmetic operations, such as division by zero.

signal handler
A subroutine or function that is called when a signal occurs.

single-byte character set (SBCS)
A coded character set in which each character is represented by a 1-byte code. A 1-byte code point allows representation of up to 256 characters. See also double-byte character set.

single precision
The use of one computer word to represent a number, in accordance with the required precision.

slash
The character /, also known as forward slash. This character is named <slash> in the portable character set.

socket
In the Network Computing System (NCS), a port on a specific host; a communications end point that is accessible through a protocol family's addressing mechanism. A socket is identified by a socket address.

software development kit (SDK)
A set of tools, APIs, and documentation to assist with the development of software in a specific computer language or for a particular operating environment.

sorted map
A sorted flat collection with key and element equality.

sorted relation
A sorted flat collection that uses keys, has element equality, and allows duplicate elements.

sorted set
A sorted flat collection with element equality.

source module
See source program.

source program
A set of instructions that are written in a programming language and must be translated into machine language before the program can be run.

space character
In the portable character set, the <space> character.

spanned record
A logical record stored in more than one block on a storage medium.

spill area
A storage area that is used to save the contents of registers.

square bracket
See bracket.

stack frame
See dynamic storage area.

standard error (STDERR)
The output stream to which error messages or diagnostic messages are sent. See also standard input, standard output.

standard input (STDIN)
An input stream from which data is retrieved. Standard input is normally associated with the keyboard, but if redirection or piping is used, the standard input can be a file or the output from a command. See also standard error.

standard output (STDOUT)
The output stream to which data is directed. Standard output is normally associated with the console, but if redirection or piping is used, the standard output can be a file or the input to a command. See also standard error.

stanza
A grouping of options in a configuration file to control various aspects of compilation by default.

statement
In programming languages, a language construct that represents a step in a sequence of actions or a set of declarations.

static binding
The act of resolving references to external variables and functions before run time.

STDERR
See standard error.

STDIN
See standard input.

STDOUT
See standard output.

storage class specifier
A storage class keyword that determines storage duration, scope, and linkage.

stream
A file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. Such objects can be created by the fdopen() or fopen() functions, and are associated with a file descriptor. A stream provides the additional services of user-selectable buffering and formatted input and output.

string
A contiguous sequence of bytes terminated by and including the first null byte.

string constant
Zero or more characters enclosed in double quotation marks. See also string literal.

string literal
Zero or more characters enclosed in double quotation marks. See also string constant.

striped data set
An extended-format data set that occupies multiple volumes. A striped data set is a software implementation of sequential data striping.

struct
See structure.

struct tag
See structure tag.

structure
A class data type that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types.

structure tag
The identifier that names a structure data type.

stub routine
Within a runtime library, a routine that contains the minimum lines of code needed to locate a given routine.

subprogram
In the IPA Link version of the Inline Report listing section, an equivalent term for 'function'.

subscript
One or more expressions, each enclosed in brackets, that follow an array name. A subscript refers to an element in an array.

subtree
A tree structure created by arbitrarily denoting a node to be the root node in a tree. A subtree is always part of a whole tree.

superset
Given two sets A and B, A is a superset of B if and only if all elements of B are also elements of A. That is, A is a superset of B if B is a subset of A.

support
In system development, to provide the necessary resources for the correct operation of a functional unit.

switch expression
The controlling expression of a switch statement.

switch statement
A C language statement that causes control to be transferred to one of several statements depending on the value of an expression.

system default
A default value defined in the system profile.

system process
An implementation-dependent object, other than a process executing an application, that has a process ID. X/Open.

T

tab character
A character that indicates that printing or displaying should start at the next horizontal position on the current line. The tab is designated by '\t' in the C language and is named in the portable character set.

text file
A file that contains only printable characters.

thread
A stream of computer instructions that is in control of a process. In some operating systems, a thread is the smallest unit of operation in a process. Several threads can run concurrently, performing different jobs.

tilde
One of the accent marks in Latin script (~).

token
The basic syntactic unit of a computing language. A token consists of one or more characters, excluding the blank character and excluding characters within a string constant or delimited identifier.

toolchain
A collection of programs or tools used to develop a product.

traceback
A section of a dump that provides information about the stack frame, the program unit address, the entry point of the routine, the statement number, and status of the routines on the call-chain at the time the traceback was produced.

trigraph
A sequence of three graphic characters that represent another graphic character. For example, in the C programming language, the trigraph ??= is used to denote the # character.

truncate
To shorten a field, value, statement, or string.

type definition
A definition of a name for a data type.

type specifier
In programming languages, a keyword used to indicate the data type of an object or function being declared.

U

ultimate consumer
The target for data in an input and output operation. An ultimate consumer can be a file, a device, or an array of bytes in memory.

ultimate producer
The source for data in an input and output operation. An ultimate producer can be a file, a device, or an array of bytes in memory.

unary expression
An expression that contains one operand.

undefined behavior
Referring to a program or function that might produce erroneous results without warning because of its use of an indeterminate value, or because of erroneous program constructs or erroneous data. See also implementation-defined.

union tag
An identifier that names a union data type.

UNIX System Services
An element of z/OS that creates a UNIX environment that conforms to XPG4 UNIX 1995 specifications and that provides two open-system interfaces on the z/OS operating system: an application programming interface (API) and an interactive shell interface.

UTC
See Coordinated Universal Time.

V

volatile attribute
An attribute of a data object that indicates the object is changeable. Any expression referring to a volatile object is evaluated immediately (for example, assignments).

W

while statement
A looping statement that executes one or more instructions repeatedly during the time that a condition is true.

white space
A sequence of one or more characters, such as the blank character, the newline character, or the tab character, that belong to the space character class.

wide character
A character whose range of values can represent distinct codes for all members of the largest extended character set specified among the supporting locales.

wide-character code
An integral value that corresponds to a single graphic symbol or control code.

wide-character string
A contiguous sequence of wide characters terminated by and including the first instance of a null wide character.

wide-oriented stream
A wide-oriented stream refers to a stream which only wide character input/output is allowed.

word
A fundamental unit of storage that refers to the amount of data that can be processed at a time. Word size is a characteristic of the computer architecture. See also doubleword, halfword.

working directory
The active directory. When a file name is specified without a directory, the current directory is searched.

writable static area (WSA)
An area of memory in a program that is modifiable during the running of a program. Typically, this area contains global variables and function and variable descriptors for dynamic link libraries (DLLs).

WSA
See writable static area.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Corporation
J74/G4
555 Bailey Avenue
San Jose, CA 95141-1099
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
3-2-12, Roppongi, Minato-ku, Tokyo 106-8711

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES
THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND,
EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF NON-INFRINGEMENT,
MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors.

Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Programming interface information

This publication documents intended Programming Interfaces that allow the customer to write Enterprise Metal C for z/OS programs.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and Trademark information (www.ibm.com/legal/copytrade.shtml).

Adobe, Acrobat, PostScript and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Standards

The following standard is supported:

- The C language is consistent with *Programming languages - C (ISO/IEC 9899:1999)* and a subset of *Programming languages - C (ISO/IEC 9899:2011)*. For more information, see International Organization for Standardization (ISO) (www.iso.org).

The following standards are supported in combination with the z/OS UNIX System Services element:

- A subset of *IEEE Std. 1003.1-2001 (Single UNIX Specification, Version 3).* For more information, see IEEE (www.ieee.org).
- *IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.
- *IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI)*, copyright 1985 by the Institute of Electrical and Electronic Engineers, Inc.
- *X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2*, copyright 1994 by The Open Group
- *X/Open CAE Specification, Networking Services, Issue 4*, copyright 1994 by The Open Group
- *X/Open Specification Programming Languages, Issue 3, Common Usage C*, copyright 1988, 1989, and 1992 by The Open Group
- United States Government's *Federal Information Processing Standard (FIPS) publication for the programming language C, FIPS-160*, issued by National Institute of Standards and Technology, 1991

Index

Special Characters

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z

Product Number: 5655-MCE

SC27-9402-00