# Using the uM-FPU64 Quaternion Instructions

Micromega Corporation

## Quaternion Register Definitions

Quaternions are stored as four element 32-bit or 64-bit register arrays, containing the w, x, y, and z quaternion values as follows:

```
q[0] = w, q[1] = x, q[2] = y, q[3] = z
```

```
qa[4]     equ    F10       ; 32-bit quaternions
qb[4]     equ    F%
qc[4]     equ    F%

qda[4]    equ    F140      ; 64-bit quaternions
qdb[4]    equ    F%
qdc[4]    equ    F%
```

## Quaternion Instructions

The uM-FPU64 quaternion instructions are implemented as `XOP`(extended opcode) instructions.

`XOP` instructions are included in a uM-FPU64 program by using the `#XOP` directive to load the `XOP` definition and `XOP` code from a library file. The `XOP` definition is used by the uM-FPU IDE to determine the number and type of arguments required, and the return type if the `XOP` is a function. The `XOP` code is stored in Flash memory along with the user defined functions.

The quaternion instructions are defined in the *quaternion.xop* library file, which is located in the *Xop Files* folder of the uM-FPU64 IDE installation folder. The following example shows the `#XOP` directive and `XOP` calls.

```
#xop quaternion:q_add
#xop quaternion:q_norm

q_add(qa, qb, qc)          ; qa = qb + qc, add quaternions
tmp = q_norm(qa)           ; calculate the norm of quaternion qa
```

## Passing Arguments to the Quaternion XOPs

`XOP` instruction are called in a similar manner to calling a procedure or function, but the argument passing method is different. Arguments are passed to `XOP` instructions by specifying a register or a pointer to a register in an 8-bit byte. If bit 7 of the byte is 0, then bits 6:0 contain the register number. If bit 7 of the byte is 1, then bits 6:0 contain the register number of a register containing a pointer a register. The uM-FPU64 IDE takes care of assigning the correct bit values based on the datatype of the argument. Quaternions are stored as register arrays, and are passed to an XOP instruction by specifying the name of the quaternion register array, or by specifying a pointer to a quaternion register array.

**Example**

```
qr[4]   equ   F10       ; rotation quaternion
qrp[4]  equ   F%        ; conjugate of rotation quaternion
qt[4]   equ   F%        ; temporary quaternion
a[3]    equ   F%        ; axis vector
v[3]    equ   F%        ; input/output vector

#function rotate(float32, @float32, @float32)
angle   equ arg1
axis    ptr arg2
vector  ptr arg3

  q_fromAngleAxis(qr, angle, axis)
  q_conjugate(qrp, qr)
  q_fromVector(qt, vector)
  q_multiply(qt, qr, qt)
  q_multiply(qt, qt, qrp)
  q_toVector(vector, qt)
#end
```

In the example above, the different methods of passing arguments are shown in the `q_fromAngleAxis XOP` call. The `qr` argument is passed as a register number since it directly references the `qr` quaternion. The `angle` is passed as a register number since it was passed to the `rotate` function as a value. The `axis` is passed as a pointer, since it was passed to the `rotate` function as a pointer.

## Code examples

The following FPU files contain examples of using the *quaternion XOP library*.

**`rotate.fp4`**

Provides functions for rotating a vector using quaternions. The *rotate* function is used to rotate a single vector. The *setRotate* and *rotateVector* would be used to rotate multiple vectors using the same rotation.

```
#function rotate(float32, @float32, @float32)
#function rotate64(float64, @float64, @float64)
    angle   equ arg1
    axis    ptr arg2
    vector  ptr arg3
```

Sets the quaternion rotation vector using the *axis* vector and *rotation* angle. Rotates the *vector* and replaces the vector with the rotated values.

```
#function setRotate(float32, @float32)
#function setRotate64(float64, @float64)
    angle   equ arg1
    axis    ptr arg2
```

Sets the quaternion rotation vector using the *axis* vector and rotation *angle*.

```
#function rotateVector(@float32)
#function rotateVector64(@float64)
    vector  ptr arg1
```

Rotates the *vector* and replaces the *vector* with the rotated values.

**`slerp_squad.fp4`**

Provides functions for slerp and squad. Additional information on these functions can be found by searching the internet.

```
#function slerp(@float32, @float32, @float32, float32)
#function slerp64(@float64, @float64, @float64, float64)
    arg1   output quaternion (qa)
    arg2   input quaternion (qb)
    arg3   input quaternion (qc)
    arg4   interpolation value (t)
```

Slerp (spherical linear interpolation), introduced by Ken Shoemake, provides quaternion interpolation. This is often used for animating 3D rotation.

```
#function squad(@float32, @float32, @float32, @float32, @float32, float32)
#function squad64(@float64, @float64, @float64, @float64, @float64, float64)
    arg1   output quaternion (qa)
    arg2   input quaternion (qb)
    arg3   input quaternion (qc)
    arg4   input quaternion (qd)
    arg5   input quaternion (qe)
    arg6   interpolation value (t)
```

Squad (spherical and quadrangle) is an interpolation curve for formulating the spherical cubic equivalent of a Bezier curve.

## Summary of 32-bit Quaternion Instructions

| | |
|---|---|
| **q_set(qa)** | Set quaternion. |
| **q_copy(qa, qb)** | Copy a quaternion. |
| **q_add(qa, qb, qc)** | Add two quaternions. |
| **q_subtract(qa, qb, qc)** | Subtract two quaternions. |
| **q_scalarMultiply(qa, qb, qc)** | Multiply quaternion by scalar. |
| **q_scalarDivide(qa, qb, qc)** | Divide quaternion by scalar. |
| **result = q_norm(qa)** | Return the norm of a quaternion. |
| **q_normalize(qa, qb)** | Normalize a quaternion. |
| **result = q_dot(qa, qb)** | Calculate dot product of two quaternions. |
| **q_conjugate(qa, qb)** | Conjugate a quaternion. |
| **q_multiply(qa, qb)** | Multiply two quaternions. |
| **q_inverse(qa, qb)** | Calculate the Inverse of a quaternion. |
| **q_fromAngleAxis(qa, angle, vb)** | Set quaternion from rotation angle and axis vector. |
| **q_fromVector(q, v)** | Set quaternion from vector. |
| **q_toVector(v, q)** | Set vector from quaternion. |
| **q_toString(q)** | Convert quaternion to a string. |

## Summary of 64-bit Quaternion Instructions

| | |
|---|---|
| **qd_set(qa)** | Set quaternion. |
| **qd_copy(qa, qb)** | Copy a quaternion. |
| **qd_add(qa, qb, qc)** | Add two quaternions. |
| **qd_subtract(qa, qb, qc)** | Subtract two quaternions. |
| **qd_scalarMultiply(qa, qb, qc)** | Multiply quaternion by scalar. |
| **qd_scalarDivide(qa, qb, qc)** | Divide quaternion by scalar. |
| **result = qd_norm(qa)** | Return the norm of a quaternion. |
| **qd_normalize(qa, qb)** | Normalize a quaternion. |
| **result = qd_dot(qa, qb)** | Calculate dot product of two quaternions. |
| **qd_conjugate(qa, qb)** | Conjugate a quaternion. |
| **qd_multiply(qa, qb)** | Multiply two quaternions. |
| **qd_inverse(qa, qb)** | Calculate the Inverse of a quaternion. |
| **qd_fromAngleAxis(qa, angle, vb)** | Set quaternion from rotation angle and axis vector. |
| **qd_fromVector(q, v)** | Set quaternion from vector. |
| **qd_toVector(v, q)** | Set vector from quaternion. |
| **qd_toString(q)** | Convert quaternion to a string. |

## Execution Times

| 32-bit Quaternions | usec |
|---|---|
| q_set | 11 |
| q_copy | 11 |
| q_add | 22 |
| q_subtract | 23 |
| q_scalarMultiply | 23 |
| q_scalarDivide | 49 |
| q_norm | 59 |
| q_normalize | 87 |
| q_dot | 44 |
| q_conjugate | 11 |
| q_multiply | 87 |
| q_inverse | 72 |
| q_fromAngleAxis | 184 |
| q_fromVector | 11 |
| q_toVector | 11 |
| q_toString | 49 |

| 64-bit Quaternions | usec |
|---|---|
| qd_set | 11 |
| qd_copy | 11 |
| qd_add | 28 |
| qd_subtract | 30 |
| qd_scalarMultiply | 43 |
| qd_scalarDivide | 131 |
| qd_norm | 105 |
| qd_normalize | 213 |
| qd_dot | 62 |
| qd_conjugate | 11 |
| qd_multiply | 158 |
| qd_inverse | 183 |
| qd_fromAngleAxis | 470 |
| qd_fromVector | 11 |
| qd_toVector | 11 |
| qd_toString | 82 |

| 32-bit Functions | usec |
|---|---|
| rotate | 431 |
| slerp | 660 |
| squad | 2395 |

| 64-bit Functions | usec |
|---|---|
| rotate | 821 |
| slerp | 1198 |
| squad | 4609 |

## 32-bit Quaternion Instruction Reference

---

**q_set** **Set quaternion.**

*XOP call:* `q_set(qa)`

*Arguments:* `qa` `float32` quaternion.

*Description:* Set quaternion `qa` to a unit identity quaternion.

```
qa(w) = 1
qa(x) = 0
qa(y) = 0
qa(z) = 0
```

---

**q_copy** **Copy a quaternion.**

*XOP call:* `q_copy(qa, qb)`

*Arguments:* `qa, qb` `float32` quaternions.

*Description:* Copy quaternion `qb` to quaternion `qa`.

```
qa(w) = qb(w)
qa(x) = qb(x)
qa(y) = qb(y)
qa(z) = qb(z)
```

---

**q_add** **Add two quaternions.**

*XOP call:* `q_add(qa, qb, qc)`

*Arguments:* `qa, qb, qc` `float32` quaternions.

*Description:* Add quaternions `qb` and `qc` and store the result in `qa`.

```
qa(w) = qb(w) + qc(w)
qa(x) = qb(x) + qc(x)
qa(y) = qb(y) + qc(y)
qa(z) = qb(z) + qc(z)
```

---

**q_subtract** **Subtract two quaternions.**

*XOP call:* `q_subtract(qa, qb, qc)`

*Arguments:* `qa, qb, qc` `float32` quaternions.

*Description:* Subtract quaternion `qc` from `qb` and store the result in `qa`.

```
qa(w) = qb(w) - qc(w)
qa(x) = qb(x) - qc(x)
qa(y) = qb(y) - qc(y)
qa(z) = qb(z) - qc(z)
```

### q_scalarMultiply — Multiply quaternion by scalar.

*XOP call:* `q_scalarMultiply(qa, qb, s)`

*Arguments:*

| | |
|---|---|
| `qa, qb` | `float32` quaternions. |
| `s` | `float32` scalar value. |

*Description:* Multiply quaternion `qb` by scalar value `s` and store the result in `qa`.

```
qa(w) = qb(w) * s
qa(x) = qb(x) * s
qa(y) = qb(y) * s
qa(z) = qb(z) * s
```

### q_scalarDivide — Divide quaternion by scalar.

*XOP call:* `q_scalarDivide(qa, qb, s)`

*Arguments:*

| | |
|---|---|
| `qa, qb` | `float32` quaternions. |
| `s` | `float32` scalar value. |

*Description:* Divide quaternion `qb` by scalar value `s` and store the result in `qa`.

```
qa(w) = qb(w) / s
qa(x) = qb(x) / s
qa(y) = qb(y) / s
qa(z) = qb(z) / s
```

### result = q_norm(qa) — Return the norm of a quaternion.

*XOP call:* `result = q_norm(qa)`

*Arguments:*

| | |
|---|---|
| `qa` | `float32` quaternion. |

*Return:*

| | |
|---|---|
| `result` | `float32` value |

*Description:* Calculate the norm of quaternion `qa` and return the value in `result`.

```
result = sqrt(qa(w)**2 + qa(x)**2 + qa(y)**2 + qa(z)**2)
```

### q_normalize — Normalize a quaternion.

*XOP call:* `q_normalize(qa, qb)`

*Arguments:*

| | |
|---|---|
| `qa, qb` | `float32` quaternions. |

*Description:* Normalize quaternion `qb` and return the result in `qa`.

```
qa[w] = qb[w] / norm(qb)
qa[x] = qb[x] / norm(qb)
qa[y] = qb[y] / norm(qb)
qa[z] = qb[z] / norm(qb)
```

**q_dot** **Calculate dot product of two quaternions.**

*XOP call:* `result = q_dot(qa, qb)`

*Arguments:* `qa, qb` `float32` quaternions.

*Description:* Calculate the dot product of quaternion `qa` and `qb`.

```
result = qa(w)*qb(w) + qa(x)*qb(x) + qa(y)*qb(y) + qa(z)*qb(z)
```

---

**q_conjugate** **Conjugate a quaternion.**

*XOP call:* `q_conjugate(qa, qb)`

*Arguments:* `qa, qb` `float32` quaternions.

*Description:* Calculate the conjugate of quaternion `qb` and return the result in `qa`.

```
qa[w] = qb[w]
qa[x] = - qb[x]
qa[y] = - qb[y]
qa[z] = - qb[z]
```

---

**q_multiply** **Multiply two quaternions.**

*XOP call:* `q_multiply(qa, qb, qc)`

*Arguments:* `qa, qb, qc` `float32` quaternions.

*Description:* Multiply quaternions `qb` and `qc` and store the result in `qa`.

```
qa(w) = qb(w)*qc(w) - qb(x)*qc(x) - qb(y)*qc(y) - qb(z)*qc(z)
qa(x) = qb(w)*qc(x) + qb(x)*qc(w) + qb(y)*qc(z) - qb(z)*qc(y)
qa(y) = qb(w)*qc(y) - qb(x)*qc(z) + qb(y)*qc(w) + qb(z)*qc(x)
qa(z) = qb(w)*qc(z) + qb(x)*qc(y) - qb(y)*qc(x) + qb(z)*qc(w)
```

---

**q_inverse** **Calculate the Inverse of a quaternion.**

*XOP call:* `q_inverse(qa, qb)`

*Arguments:* `qa, qb` `float32` quaternions.

*Description:* Calculate the inverse of quaternion `qb` and store the result in `qa`.

```
qa = conjugate(qb) / (norm(qb) ** 2)
qa = qb(w) / (norm(qb) ** 2)
qa = -qb(x) / (norm(qb) ** 2)
qa = -qb(y) / (norm(qb) ** 2)
qa = -qb(z) / (norm(qb) ** 2)
```

---

**q_fromAngleAxis** **Set quaternion from rotation angle and axis vector.**

*XOP call:* `q_fromAngleAxis(qa, angle, axis)`

*Arguments:* `qa` `float32` quaternion.

| | | |
|---|---|---|
| | `angle` | `float32` value (in radians). |
| | `axis` | `float32` vector. |

*Description:* Sets quaternion `qa` from the `angle` value and the vector `axis`.

---

**q_fromVector** **Set quaternion from vector.**

*XOP call:* `q_fromVector(`*`qa`*`, `*`v`*`)`

| | | |
|---|---|---|
| *Arguments:* | `qa` | `float32` quaternion. |
| | `v` | `float32` vector. |

*Description:* Set quaternion `qa` from vector `v`.

```
q(w) = 0
q(x) = v(x)
q(y) = v(y)
q(z) = v(z)
```

---

**q_toVector(v, q)** **Set vector from quaternion.**

*XOP call:* `q_toVector(v, `*`qa`*`)`

| | | |
|---|---|---|
| *Arguments:* | `v` | `float32` vector. |
| | `qa` | `float32` quaternion. |

*Description:* Set vector `v` from quaternion `qa`.

```
v(x) = q(x)
v(y) = q(y)
v(z) = q(z)
```

---

**q_toString(q)** **Convert quaternion to a string.**

*XOP call:* `q_toString(`*`qa`*`)`

| | | |
|---|---|---|
| *Arguments:* | `qa` | `float32` quaternion. |

*Description:* Convert quaternion `qa` to a string and store at the string in the string buffer at the current selection point. e.g “(1, 0, 0, 0)”

---

## 64-bit Quaternion Instruction Reference

---

**qd_set** **Set quaternion (64-bit).**

*XOP call:* `qd_set(qa)`

*Arguments:* `qa` `float64` quaternion.

*Description:* Set quaternion `qa` to a unit identity quaternion.

```
qa(w) = 1
qa(x) = 0
qa(y) = 0
qa(z) = 0
```

---

**qd_copy** **Copy a quaternion (64-bit).**

*XOP call:* `qd_copy(qa, qb)`

*Arguments:* `qa, qb` `float64` quaternions.

*Description:* Copy quaternion `qb` to quaternion `qa`.

```
qa(w) = qb(w)
qa(x) = qb(x)
qa(y) = qb(y)
qa(z) = qb(z)
```

---

**qd_add** **Add two quaternions (64-bit).**

*XOP call:* `qd_add(qa, qb, qc)`

*Arguments:* `qa, qb, qc` `float64` quaternions.

*Description:* Add quaternions `qb` and `qc` and store the result in `qa`.

```
qa(w) = qb(w) + qc(w)
qa(x) = qb(x) + qc(x)
qa(y) = qb(y) + qc(y)
qa(z) = qb(z) + qc(z)
```

---

**qd_subtract** **Subtract two quaternions (64-bit).**

*XOP call:* `qd_subtract(qa, qb, qc)`

*Arguments:* `qa, qb, qc` `float64` quaternions.

*Description:* Subtract quaternion `qc` from `qb` and store the result in `qa`.

```
qa(w) = qb(w) - qc(w)
qa(x) = qb(x) - qc(x)
qa(y) = qb(y) - qc(y)
qa(z) = qb(z) - qc(z)
```

### qd_scalarMultiply — Multiply quaternion by scalar (64-bit).

*XOP call:* `qd_scalarMultiply(qa, qb, s)`

| | | |
|---|---|---|
| *Arguments:* | `qa, qb` | `float64` quaternions. |
| | `s` | `float64` scalar value. |

*Description:* Multiply quaternion `qb` by scalar value `s` and store the result in `qa`.

```
qa(w) = qb(w) * s
qa(x) = qb(x) * s
qa(y) = qb(y) * s
qa(z) = qb(z) * s
```

### qd_scalarDivide — Divide quaternion by scalar (64-bit).

*XOP call:* `qd_scalarDivide(qa, qb, s)`

| | | |
|---|---|---|
| *Arguments:* | `qa, qb` | `float64` quaternions. |
| | `s` | `float64` scalar value. |

*Description:* Divide quaternion `qb` by scalar value `s` and store the result in `qa`.

```
qa(w) = qb(w) / s
qa(x) = qb(x) / s
qa(y) = qb(y) / s
qa(z) = qb(z) / s
```

### result = qd_norm(qa) — Return the norm of a quaternion (64-bit).

*XOP call:* `result = qd_norm(qa)`

| | | |
|---|---|---|
| *Arguments:* | `qa` | `float64` quaternion. |
| *Return:* | `result` | `float64` value |

*Description:* Calculate the norm of quaternion `qa` and return the value in `result`.

```
result = sqrt(qa(w)**2 + qa(x)**2 + qa(y)**2 + qa(z)**2)
```

### qd_normalize — Normalize a quaternion (64-bit).

*XOP call:* `qd_normalize(qa, qb)`

| | | |
|---|---|---|
| *Arguments:* | `qa, qb` | `float64` quaternions. |

*Description:* Normalize quaternion `qb` and return the result in `qa`.

```
qa[w] = qb[w] / norm(qb)
qa[x] = qb[x] / norm(qb)
qa[y] = qb[y] / norm(qb)
qa[z] = qb[z] / norm(qb)
```

**qd_dot** **Calculate dot product of two quaternions (64-bit).**

*XOP call:* `result = q_dot(qa, qb)`

*Arguments:* `qa, qb` `float64` quaternions.

*Description:* Calculate the dot product of quaternion `qa` and `qb`.

```
result = qa(w)*qb(w) + qa(x)*qb(x) + qa(y)*qb(y) + qa(z)*qb(z)
```

---

**qd_conjugate** **Conjugate a quaternion (64-bit).**

*XOP call:* `qd_conjugate(qa, qb)`

*Arguments:* `qa, qb` `float64` quaternions.

*Description:* Calculate the conjugate of quaternion `qb` and return the result in `qa`.

```
qa[w] = qb[w]
qa[x] = - qb[x]
qa[y] = - qb[y]
qa[z] = - qb[z]
```

---

**qd_multiply** **Multiply two quaternions (64-bit).**

*XOP call:* `qd_multiply(qa, qb, qc)`

*Arguments:* `qa, qb, qc` `float64` quaternions.

*Description:* Multiply quaternions `qb` and `qc` and store the result in `qa`.

```
qa(w) = qb(w)*qc(w) - qb(x)*qc(x) - qb(y)*qc(y) - qb(z)*qc(z)
qa(x) = qb(w)*qc(x) + qb(x)*qc(w) + qb(y)*qc(z) - qb(z)*qc(y)
qa(y) = qb(w)*qc(y) - qb(x)*qc(z) + qb(y)*qc(w) + qb(z)*qc(x)
qa(z) = qb(w)*qc(z) + qb(x)*qc(y) - qb(y)*qc(x) + qb(z)*qc(w)
```

---

**qd_inverse** **Calculate the Inverse of a quaternion (64-bit).**

*XOP call:* `qd_inverse(qa, qb)`

*Arguments:* `qa, qb` `float64` quaternions.

*Description:* Calculate the inverse of quaternion `qb` and store the result in `qa`.

```
qa = conjugate(qb) / (norm(qb) ** 2)
qa = qb(w) / (norm(qb) ** 2)
qa = -qb(x) / (norm(qb) ** 2)
qa = -qb(y) / (norm(qb) ** 2)
qa = -qb(z) / (norm(qb) ** 2)
```

---

**qd_fromAngleAxis** **Set quaternion from rotation angle and axis vector (64-bit).**

*XOP call:* `qd_fromAngleAxis(qa, angle, axis)`

*Arguments:* `qa` `float64` quaternion.
`angle` `float64` value (in radians).
`axis` `float64` vector.

*Description:* Sets quaternion `qa` from the `angle` value and the vector `axis`.

---

**qd_fromVector** **Set quaternion from vector (64-bit).**

*XOP call:* `qd_fromVector(`*`qa`*`, `*`v`*`)`

*Arguments:* `qa` `float64` quaternion.
`v` `float64` vector.

*Description:* Set quaternion `qa` from vector `v`.

```
q(w) = 0
q(x) = v(x)
q(y) = v(y)
q(z) = v(z)
```

---

**qd_toVector(v, q)** **Set vector from quaternion (64-bit).**

*XOP call:* `qd_toVector(v, `*`qa`*`)`

*Arguments:* `v` `float64` vector.
`qa` `float64` quaternion.

*Description:* Set vector `v` from quaternion `qa`.

```
v(x) = q(x)
v(y) = q(y)
v(z) = q(z)
```

---

**qd_toString(q)** **Convert quaternion to a string (64-bit).**

*XOP call:* `qd_toString(`*`qa`*`)`

*Arguments:* `qa` `float64` quaternion.

*Description:* Convert quaternion `qa` to a string and store at the string in the string buffer at the current selection point. e.g “(1, 0, 0, 0)”

---

# Further Information

See the Micromega website (http://www.micromegacorp.com) for additional information regarding the uM-FPU64 floating point coprocessor, including:

*uM-FPU64 Datasheet*
*uM-FPU64 Instruction Set*