

---

# Consensus

**Asynchronous lock-free Three-Buffer Data-Sharing  
for Embedded System using C/C++  
Technical Concept Paper**

---



**carrara engineering GmbH**  
Reto Carrara  
Gyrenstrasse 17  
CH-8967 Widen  
+4179/328'17'05  
[www.carrara.ch](http://www.carrara.ch)  
[info@carrara.ch](mailto:info@carrara.ch)

# Inhalt

Inhalt.....	2
Revision.....	3
Einführung.....	4
Quelle.....	4
Reader-/Writer-Problematik.....	4
Nicht-blockierender Datenaustausch.....	5
Consensus.....	7
Problematik.....	7
TestAndSet.....	7
Die Konsensfindung.....	8
Volatile.....	9
Asynchrone Drei-Buffer Kommunikation.....	10
Übersicht.....	10
Szenario.....	10
Permutation des Write-Buffers.....	11
Was ist der Konsens.....	12
Anwendung des Consensus Algorithmus.....	13
Globale Variablen.....	13
Reader.....	13
Writer.....	13
Implementation des Consensus Algorithmus.....	15
WriteConsensus.....	15
ReaderConsensus.....	16
Gesamtlösung.....	17
Initialisierung.....	18
Test.....	19
Multiprozessor Betriebssysteme.....	20
Cache Problematik.....	20
Multiprozessor Windows auf X86.....	20
Multiprocessor VxWorks auf X86.....	21
Verifikation unter Multiprozessorbetriebssystemen.....	21
C++ Implementation.....	23
Einführung.....	23
Shared Data.....	23
Buffer.....	23
ReadConsensus.....	24
WriteConsensus.....	25
Anwendung.....	25
Schlussüberlegungen.....	27
Literatur.....	28

## Revision

---

<b>Datum</b>	<b>Version</b>	<b>Autor</b>	<b>Änderungen</b>
30.01.04	1.0	rc	Basisversion
01.02.04	1.1	sf	Korrektur
02.02.04	1.2	rc	Korrektur
03.02.04	1.3	rc	Korrektur in Consens5::WriteConsensus und Doc
16.02.04	1.4	rc	Korrektur: Assembler Pseudocode
23.04.04	1.5	rc	long& anstelle bool& bei TestAndSet, da bool nur 1 Byte; vxTas für portable TestAndSet-Lösung unter Vx
29.04.04	1.6	rc	Korrektur für VxTas TestAndSet Impl

## Einführung

### Quelle

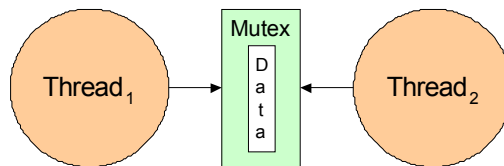
Dieser Artikel basiert auf der bahnbrechenden Arbeit von *Jing Chen* und *Alan Burns* von der *Real Time System Group* der *University of York, U.K.*

Der Titel ihres Papers<sup>1</sup> lautet:

Asynchronous Data Sharing in Multiprocessor Real-Time Systems Using Process Consensus

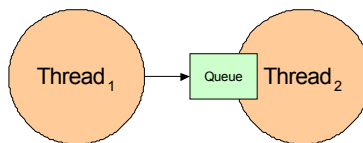
### Reader-/Writer-Problematik

Gerade in Echtzeitsystemen wird häufig mit Threads<sup>2</sup> gearbeitet, die untereinander Daten austauschen müssen. Um aus mehreren Threads sicher auf Datenbestände zugreifen zu können, müssen diese durch eine Mutex-Section geschützt werden.



Diese Methode hat sich als fehleranfällig erwiesen, da Programmierer häufig die korrekte Handhabung unterlassen<sup>3</sup>. Es ist daher oft einfacher, eine Multithreading-Anwendung als reaktives Meldungssystem umzusetzen.

In einem Meldungssystem erfolgt der Datenaustausch zwischen den einzelnen Threads - wie der Name schon sagt - vollständig mit Hilfe von Meldungen. Hierzu verfügt jeder Thread über eine eigene Meldungsqueue. Befindet sich keine Meldung in der Queue, so legt sich der Thread schlafen.



Zwei Dinge erweisen sich bei dieser Lösung als problematisch: Erstens muss Thread<sub>1</sub> jeweils eine Meldung erzeugen, die von Thread<sub>2</sub> gelesen und anschliessend gelöscht werden muss. Diese Operationen kosten Zeit, da hierzu der Kernel<sup>4</sup> bemüht werden muss. Zweitens braucht auch die

<sup>1</sup> Siehe Literatur [9]

<sup>2</sup> Threads sind parallele Ausführungseinheiten innerhalb einer Anwendung (Prozess), die direkt über Speicherzugriffe miteinander kommunizieren können. Da alle Threads eines Prozesses Zugriff auf dieselben Heapdaten haben, können sie Daten untereinander schnell und effizient austauschen.

<sup>3</sup> Als Beispiel sei hier ein Programmierer genannt, der für jeden Prozess eine neue Mutex-Semaphore erzeugt hat, um seine Daten zu schützen.

<sup>4</sup> Selbst wenn ein eigenes Memory Management eingesetzt wird, ist eine Mutex-Semaphore nötig, um alloc und free sicher zu implementieren.

gegen Über- und Unterlauf synchronisierte Meldungsqueue zwei Counting- und eine Mutex-Sempahore.

Interessant wäre darum ein Datenaustausch zwischen Threads, der möglichst wenig oder gar keine Kernelinteraktion benötigt, aber trotzdem sicher ist.

## Nicht-blockierender Datenaustausch

Für einen Datenaustausch ohne Kernelinteraktion müssen gewisse Vorraussetzung geben sein:

- Der Writer<sup>5</sup> schreibt seine Daten sobald sie zur Verfügung stehen in einen Buffer
- Der Reader<sup>6</sup> liest Daten sobald sie benötigt werden von einem Buffer

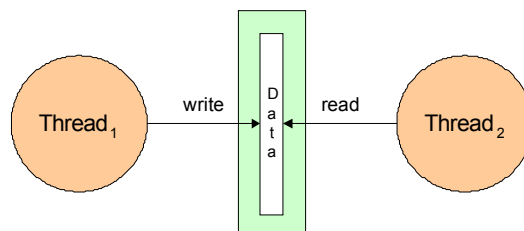
Diese Form des Datenaustausches unterscheidet sich wesentlich von einem Messagequeue-basierten Datenaustausch. So kann der Writer nie sicher sein, dass der Reader die anstehenden Daten schnell genug liest, um jede Änderung mitzubekommen. Das heisst, wenn der Writer viel schneller als der Reader ist, kann nicht jeder geschriebene Buffer auch gelesen werden.

Auch im umgekehrten Fall kommt es zu Problemen: Wenn der Reader schneller liest als der Writer schreibt, so erhält der Reader solange dieselben Daten, bis neue zur Verfügung stehen.

Diese Probleme gilt es zu lösen, damit ein nicht-blockierender Datenaustausch sinnvoll möglich ist. Doch wenden wir uns zuerst der Frage zu, wofür ein solches System überhaupt eingesetzt werden kann. Aufträge können so schliesslich nur schlecht erteilt werden, da der Reader per Definition nicht notwendigerweise alle Änderungen mitkriegt.

Wir denken, dass sich die oben beschriebene Algorithmik sehr gut für den Austausch von Konfigurationsdaten eignet, so zum Beispiel die Arbeitsgeschwindigkeit eines Roboters.

Konfigurationsdaten ändern von Zeit zu Zeit, aber der Empfänger der Daten kann sowohl mit den alten als auch mit den neuen Daten umgehen. Wichtig ist dabei nur, dass immer auf möglichst aktuellen Daten gearbeitet werden kann.



Das Ziel von Reader und Writer besteht demnach darin, sich zu einigen, welches die neusten Daten sind. Kann diese Einigung in kürzester Zeit erreicht werden, und zwar ohne vom Betriebssystem blockiert zu werden, so können sowohl der Writer als auch der Reader unabhängig<sup>7</sup> voneinander ihrer Arbeit nachgehen.

<sup>5</sup> Als Writer wird derjenige Thread bezeichnet, welcher Daten produziert und in einen Buffer schreibt.

<sup>6</sup> Als Reader wird derjenige Thread bezeichnet, welcher Daten aus einem Buffer liest und anschliessend konsumiert.

<sup>7</sup> Ohne Betriebssystemsynchrisation über binäre Sempahoren



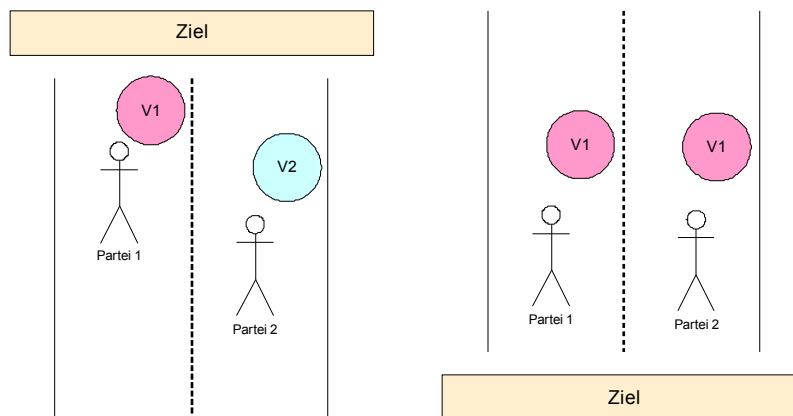
# Consensus

## Problematik

Um das beschriebene Problem angehen zu können, müssen wir uns zuerst mit der sogenannten Konsensfindung zwischen zwei Parteien beschäftigen. Dabei geht es darum, dass parallele Entitäten sicher zu einer Einigung kommen

Beispiel:

Zwei Parteien müssen einen Vorschlag  $V$  einreichen. Diejenige Partei, welche den Vorschlag  $V_x$  zuerst eingereicht hat, ist die Gewinnerin. Sowohl die Gewinner- als auch die Verlierer-Partei müssen den Vorschlag der Gewinner-Partei annehmen.



## TestAndSet

Damit wir uns an die Lösung der obigen Problemstellung machen können, muss zuerst das Konzept TestAndSet eingeführt werden.

TestAndSet ermöglicht es, eine Speicherstelle **atomar**<sup>8</sup> auszulesen, mit dem Wert 1 zu beschreiben und den alte Wert zu retournieren. In C implementiert sieht der Code folgendermassen aus:

```
bool testAndSet(bool& i_raFlag)
{
    // Start Atomic
    bool bFlag = i_raFlag;
    i_raFlag = true;
    return bFlag;
    // End Atomic
}
```

Da dieser Befehl für die Synchronisation in einem Multithreading-Umfeld sehr wichtig ist, wird er von den meisten heutigen Prozessoren in der einen oder anderen Form zur Verfügung gestellt.

<sup>8</sup> D.h. ohne von einem anderen Prozess oder einem Interrupt unterbrochen zu werden.

Auf einer X86-Architektur kann TestAndSet mittels xchg (Exchange) realisiert werden. xchg tauscht dabei der Wert einer Speicherstelle durch den Wert in einem Register aus, wobei der Wert der Speicherstelle nach der Operation im Register zu liegen kommt.

Pseudocode:

```
mov r1, @val
mov r2, 1
xchg [r1], r2
```

Unter Windows NT/XP existiert der Befehl InterlockedExchange, welcher ein xchg ermöglicht.

```
LONG InterlockedExchange(
    LONG volatile* Target,
    LONG Value
);
```

Unter dem Betriebssystem Windows könnte die TestAndSet-Funktion als folgendermassen implementiert werden:

```
bool testAndSet(volatile long& i_raFlag)
{
    return InterlockedExchange(&i_raFlag, true)?true:false;
}
```

## Die Konsensfindung

Mit Hilfe von TestAndSet ist es nun möglich, dass wir uns an das Problem der Konsensfindung wagen:

```
volatile T_Type s_aPreference[2];
volatile long s_bSync = false;

T_Type getP1Consensus(const T_Type& i_raValue)
{
    s_aPreference[0] = i_raValue;
    if (testAndSet(s_bSync)==false)
    {
        return s_aPreference[0];
    }
    else
    {
        return s_aPreference[1];
    }
}

T_Type getP2Consensus(const T_Type& i_raValue)
{
    s_aPreference[1] = i_raValue;
    if (testAndSet(s_bSync)==false)
    {
        return s_aPreference[1];
    }
    else
    {
        return s_aPreference[0];
    }
}
```

Wir erinnern uns, dass die Daten des ersten Prozesses sowohl vom ersten als auch von den weiteren Prozessen übernommen werden sollen.

```
L01 T_Type getP1Consensus(const T_Type& i_raValue)
L02 {
L03     s_aPreference[0] = i_raValue;
L04     if (testAndSet(s_bSync)==false)
L05     {
L06         return s_aPreference[0];
L07     }
L08     else
L09     {
L10         return s_aPreference[1];
L11     }
L12 }
```

Aus diesem Grund notiert sich jeder Prozess seine Präferenz, das heisst seine eigenen Daten (L03). Als nächstes wird mit TestAndSet ermittelt, ob bereits ein anderer Prozess seinen Vorschlag eingereicht hat oder nicht. Ist dies nicht der Fall, so liefert TestAndSet den Wert 0 und setzt das globale Sync-Flag auf 1. Hat ein Prozess das "Rennen" gewonnen, so werden als Resultat die eigenen Daten retourniert. War ein anderer Prozess schneller, werden dessen Daten retourniert.

Es gilt zu beachten, dass obiger Algorithmus keine Race-Conditions<sup>9</sup> zulässt. Ist Prozess<sub>1</sub> bei L03 angelangt, kann ihm der Prozessor durch einen Interrupt<sup>10</sup> entzogen werden. In der Zwischenzeit kann der Prozess<sub>2</sub> sowohl L03 als auch L04 ausführen. Da er das Ziel (die Methode TestAndSet) als erster erreicht hat, wird sein Resultat bevorzugt. Sprich, das globale Sync-Flag wird durch TestAndSet auf 1 gesetzt und der Prozess<sub>1</sub> erhält dieses Resultat, nachdem er vom Betriebssystem wieder geweckt wurde.

Im obigen Beispiel haben wir nun die Grundlagen für eine Lock- und Loop-freie Datenkommunikation erarbeitet. Wir werden im Folgenden zeigen, wie diese umgesetzt werden kann.

## Volatile

Da auf die globalen Variablen `s_aPreference` und `s_bSync` von mehr als einem Thread zugegriffen wird, müssen diese als `volatile` deklariert werden. Damit wird sichergestellt, dass der Compiler die Werte immer aus dem Speicher holt und keine Optimierungsstrategien über Register anwendet.

<sup>9</sup> Man spricht von einer Race-Condition, wenn das Resultat davon abhängt, ob ein Prozess schneller auf gemeinsame Ressourcen zugreift als ein anderer.

<sup>10</sup> In Echtzeitsystemen ist es eigentlich nie empfehlenswert, einen Scheduler zu verwenden. Vielmehr soll ein sauberes reaktives System aufgebaut werden. Trotzdem kann einem Prozess/Thread der Prozessor entzogen werden, wenn ein Interrupt abgehandelt werden muss. Welcher Prozess/Thread nach Behandlung des Interrupts an die Reihe kommt ist Sache des Kernels.

# Asynchrone Drei-Buffer Kommunikation

## Übersicht

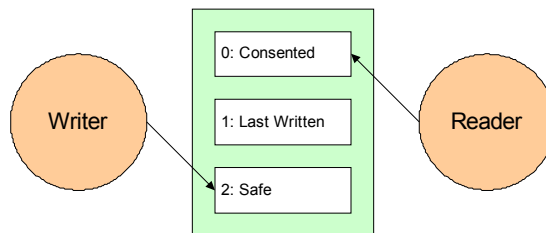
Damit zwei Prozesse ohne Lock miteinander kommunizieren können, müssen drei Buffer zur Verfügung stehen:

1. Der Buffer, aus dem gelesen wird
2. Der Buffer, der die nächsten aktuellen Daten beinhaltet
3. Der Buffer, in den geschrieben werden kann

Für die Buffer sollen prägnante Namen gefunden werden:

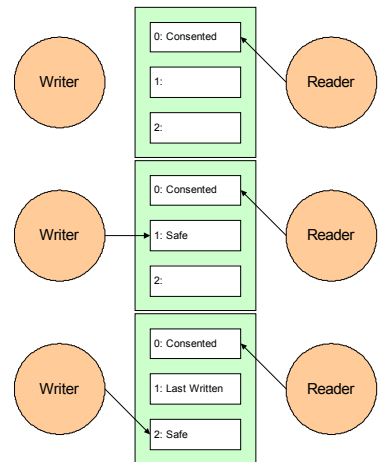
Consented Buffer	Sowohl der Reader als auch der Writer haben sich geeinigt, welcher Buffer zum Lesen zur Verfügung steht: es wurde ein Konsens erreicht.
Last Written Buffer	In diesen Buffer wurde vom Writer zuletzt geschrieben.
Safe Buffer	In diesen Buffer kann der Writer jederzeit schreiben. [SafeWriteBuffer]

Somit können wir festhalten, dass der Writer immer in den Safe Buffer zu schreiben hat, während der Reader immer vom Consented Buffer lesen muss.

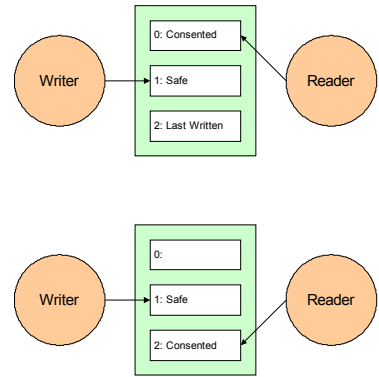


## Szenario

- 1 Der Consented Buffer ist an Platz #0. Der Wert ist bereits vorinitialisiert, so dass der Reader jederzeit lesen kann.
- 2 Der Writer beschliesst, Daten zu schreiben. Buffer #1 wird zum Safe Buffer.
- 3 Der Writer hat das Schreiben der Daten abgeschlossen und gibt diese zur Nutzung frei. Dadurch wird #1 zum Last Written und #2 zum Safe Buffer.



- 4 Nehmen wir nun an, der Reader so sehr beschäftigt, dass der Writer bereits wieder neue Daten geschrieben hat. Dies bedeutet, dass der Buffer #2 neu zum Last Written und Buffer #1 wiederum zum Safe Buffer wird, da der Reader immer noch von #0 liest. Damit sind die ehemals neuen Daten im Buffer #1 hinfällig geworden.
- 5 Nun sei auch der Reader an einem Punkt angelangt, an dem neue Daten benötigt werden. Der Reader und der Writer müssen sich in einer Konsensbildung auf einer neuen Buffer einigen. Im Falle dass der Reader nun schneller als der Writer ist, muss der Reader trotzdem weiterhin von Buffer #2 lesen, da noch kein neu geschriebener Buffer zur Verfügung steht.



### Permutation des Write-Buffers

Wir können aus dem obigen Szenario folgendes ableiten:

- Ist der Reader schneller als der Writer, so liest er solange dieselben Daten, bis neue zur Verfügung stehen.
- Ist der Writer schneller als der Reader, so müssen die neuen Daten abwechslungsweise in einen der beiden Buffer geschrieben werden, aus denen nicht gelesen wird.
- Hat der Reader die neuen Daten übernommen, steht es dem Writer frei, in welchen der beiden Buffer (aus denen *nicht* gelesen wird) geschrieben werden soll.

Aufgrund der Information, welches der Consented Buffer und welches der Last Written Buffer ist, lässt sich der Safe Buffer bestimmen.

Last Written		Consented Buffer		
		0	1	2
0	1 oder 2	2	1	
1	2	0 oder 2	0	
2	1	0	0 oder 1	

Wenn wir uns für die Fälle in der Diagonale willkürlich festlegen, lässt sich eine eindeutige Zuordnung vornehmen:

1	2	1
2	2	0
1	0	0

Damit die obige Entscheidung sehr schnell getroffen werden kann, wird eine Permutationstabelle angelegt, welche folgende Funktion erfüllt:

```
f_SaveBufferIdx(ConsentedBufferIndex, LastWrittenBufferIndex)
```

```
const int nPermutatorTab[3][3] = {{1,2,1},{2,2,0},{1,0,0}};
```

## **Was ist der Konsens**

Bevor wir nun näher auf die Anwendung eingehen, müssen wir uns darüber im Klaren sein, wofür ein Konsens gefunden werden muss.

Bei näherer Betrachtung fällt uns auf, dass der Reader immer vom Consented Buffer liest. Der Writer hingegen schreibt weder in den Consented Buffer noch in denjenigen Buffer, in den er zuletzt geschrieben hat.

Alleine der Writer muss wissen, in welchen Buffer zuletzt geschrieben wurde. Hingegen müssen sowohl Reader als Writer wissen, welches der Consented Buffer ist. Wie der Name schon sagt soll der Konsens für den Index des Consented Buffers gefunden werden.

## Anwendung des Consensus Algorithmus

Ohne auf die genauere Implementation von Read- und WriteConsensus einzugehen, soll deren Anwendung aufgezeigt werden.

### Globale Variablen

Folgende globalen Variablen sind für die Anwendung entscheidend:

```
volatile T_Type s_aBuffer[3];
volatile int    s_nLastWritten;
```

### Reader

Vom ReaderConsensus wird erwartet, dass der Index des Consented Buffers zurückgegeben wird. Somit funktioniert das Lesen folgendermassen:

```
void testReader()
{
    int nConsented = getReadConsensus();
    cout << "Read[" << nConsented << "]: " << s_aBuffer[nConsented] << endl;
}
```

Wichtig: Der Buffer-Index nConsented kann solange verwendet werden, wie kein neuer Konsens angestrebt wird<sup>11</sup>.

### Writer

Vom WriterConsensus wird ebenfalls erwartet, dass der Index des Consented Buffers zurückgegeben wird. Für die korrekte Permutation der Buffer wird zudem der Index des zuletzt beschriebenen Buffers benötigt; dies für den Fall, dass der Writer schneller als der Reader ist.

Für die Bestimmung des Consented Buffer im Rahmen der Konsensfindung benötigen sowohl der Reader als auch der Writer den Index des zuletzt geschriebenen Buffers. Daher wird diese Variable global verwaltet.

```
void testWriter(int i_nData)
{
    static const int nPermutator[3][3]
        = {{1,2,1}
          ,{2,2,0}
          ,{1,0,0}};

    int nConsented = getWriteConsensus();
    int nSafe      = nPermutator[nConsented][s_nLastWritten];
    s_aBuffer[nSafe] = i_nData;
    s_nLastWritten  = nSafe;
}
```

<sup>11</sup> Dies wird über den erneuten Aufruf von getReadConsensus erreicht.

Als erstes legen wir im Writer unsere Tabelle an, welche die Buffer-Permutation ermöglicht. Daraufhin kann ein WriteConsensus getroffen werden. Als Resultat erwarten wir - wie bereits erwähnt - den Consented Buffer.

Mittels des globalen Index Last Written kann der Index des Safe Buffers bestimmt werden, in welchen dann geschrieben werden kann. Zum Schluss wird der Last Written Index auf den soeben bearbeiteten Buffer gesetzt.

Wichtig: Der Schreibvorgang kann solange fortgesetzt werden wie kein Commit ausgelöst wurde. Ein Commit wird erreicht, in dem der Index des zuletzt geschriebenen Buffers neu gesetzt wird.

## Implementation des Consensus Algorithmus

Zum Schluss soll gezeigt werden, wie mit drei Buffern ein asynchrones Data-Sharing mittels TestAndSet erreicht werden kann. Dazu wird zuerst der WriteConsensus entwickelt.

### WriteConsensus

Grundsätzlich unterscheiden sich Read- und WriteConsensus kaum von der Standard Consensus Implementation.

```
// Global State
volatile T_Type s_aBuffer[3];
volatile long s_bSync;
volatile int s_nLastWritten;
volatile int s_nReaderPref;
volatile int s_nWriterPref;
```

```
L01 int getWriteConsensus()
L02 {
L03     s_nWriterPref = s_nLastWritten;
L04     if (testAndSet(s_bSync)==false)
L05     {
L06         return s_nWriterPref;
L07     }
L08     else
L09     {
L10         return s_nReaderPref;
L11     }
L12 }
```

Der Writer testet, in welchen Buffer zuletzt geschrieben wurde. Dieser Index wird als bevorzugte Antwort für den WriteConsensus gesetzt (L03). Über TestAndSet (L04) wird geprüft, ob der Writer vor dem Reader aufgerufen wurde. Wenn ja, wird als Resultat der bevorzugte Index des Writers als Konsens festgelegt. Andernfalls macht der Reader das Rennen.

Was geschieht nun aber, wenn der Writer jeweils schneller als der Reader ist? Die Anforderung an den WriteConsensus, welche aus dem vorgängige beschriebenen Anwendungsszenario entstanden ist, besagt, dass als Resultat der Consented Buffer erwartet wird. Der Writer selber aber setzt den Last Written Index jeweils neu, um seine neuen Daten zu bestätigen. Aus diesem Grund müssen wir uns den letzten Buffer merken, für welchen ein Konsens gefunden wurde. Dieser 'alte' Buffer-Index wird immer dann zurückgegeben, wenn der Reader noch nicht neu gelesen hat. Um diese Information zu erhalten, wird ein Read Signal eingeführt.

```
// Global State
volatile T_Type s_aBuffer[3];
volatile long s_bSync;
volatile bool s_bReadSignal;
volatile int s_nLastWritten;
volatile int s_nReaderPref;
volatile int s_nWriterPref;
```

Zusätzlich braucht der WriteConsensus den Status `s_nConsented`.

```
// Writers State
int s_nConsented;

L01 int getWriteConsensus()
L02 {
L03     if (s_bReadSignal)
L04     {
L05         s_bReadSignal = false;
L06
L07         s_nWriterPref = s_nLastWritten;
L08         if (testAndSet(s_bSync)==false)
L09         {
L10             s_nConsented = s_nWriterPref;
L11         }
L12     else
L13     {
L14         s_nConsented = s_nReaderPref;
L15     }
L16 }
L17 return s_nConsented;
L18 }
```

Auf L03 wird bestimmt, ob der Reader bereits neue Daten angefordert hat. Wenn ja, wird das Flag für den nächsten Durchlauf zurückgesetzt (L05) und der zuletzt geschriebene Buffer auf L07 als Präferenz gewählt<sup>12</sup>. Über TestAndSet wird dann entschieden, welche Präferenz der Variablen `s_nConsented` zugewiesen wird.

## ReaderConsensus

Ist der Writer erst einmal verstanden, so können wir uns schliesslich an den Reader wagen.

```
L01 int getReadConsensus()
L02 {
L03     s_bSync          = false;
L04     s_bReadSignal    = true;
L05
L06     s_nReaderPref    = s_nLastWritten;
L07     if (testAndSet(s_bSync)==false)
L08     {
L09         return s_nReaderPref;
L10     }
L11     else
L12     {
L13         return s_nWriterPref;
L14     }
L15 }
```

Natürlich funktioniert auch der Reader nach dem mittlerweile bekannten Muster zur Konsensfindung. Da der Reader mit seiner Geschwindigkeit bestimmt, ob ein neuer Konsens gefunden werden muss, setzt er das Sync Flag jeweils auf false (L03). Zusätzlich muss natürlich auch signalisiert werden, dass ein neuer Buffer gelesen wird, so dass der Writer aus seinem

<sup>12</sup> Der Reader versucht übrigens dasselbe zu erreichen. Dank TestAndSet kann eine Race-Condition ausgeschlossen werden.

Buffer-Permutations-Mode "ausbrechen" kann. Dies wird über das Setzen des Read Signals auf L04 erreicht.

## Gesamtlösung

Es sei an dieser Stelle noch einmal daran erinnert, dass alle Variablen, die von mehr als einem Thread gelesen und geschrieben werden sollen als `volatile` deklariert werden müssen.

```
// Global State
volatile T_Type s_aBuffer[3];
volatile long   s_bSync;
volatile bool   s_bReadSignal;
volatile int    s_nLastWritten;
volatile int    s_nReaderPref;
volatile int    s_nWriterPref;
```

Der Status der Writers umfasst den letzten Buffer Index, für den ein Konsens gefunden wurde. Dadurch wird es möglich, die Buffer abwechslungsweise zu beschreiben, falls der Reader langsamer ist.

```
// Writers State
int s_nConsented;
```

Der WriteConsensus ermittelt den zuletzt getroffenen Konsens, das heisst den Index des Consented Buffers. Falls ReadConsensus zwischenzeitlich nicht aufgerufen wurde, wird das Read Signal nicht gesetzt, woraufhin der Writer den zuletzt getroffenen Konsens aus der Temporär-Variablen `s_nConsented` ermittelt.

```
int getWriteConsensus()
{
    if (s_bReadSignal)
    {
        s_bReadSignal = false;

        s_nWriterPref = s_nLastWritten;
        if (testAndSet(s_bSync)==false)
        {
            s_nConsented = s_nWriterPref;
        }
        else
        {
            s_nConsented = s_nReaderPref;
        }
    }
    return s_nConsented;
}
```

Der Reader ist für das Initiieren einer Konsensfindung zuständig. Aus diesem Grund setzt er das Sync Flag `s_bSync` jeweils zu Beginn der Konsensfindung zurück. Des Weiteren signalisiert er dem Writer mittels Read Signal seine Bereitschaft zu lesen.

```
int getReadConsensus()
{
```

```

s_bSync      = false;
s_bReadSignal = true;

s_nReaderPref = s_nLastWritten;
if (testAndSet(s_bSync)==false)
{
    return s_nReaderPref;
}
else
{
    return s_nWriterPref;
}
}

```

Der Reader erhält als Resultat des `ReadConsensus` den Index des Consented Buffers. Dieser Index ist bis zur nächsten Konsensfindung gültig.

```

void testReader()
{
    int nConsented = getReadConsensus();
    cout << "Read[" << nConsented << "]: " << s_aBuffer[nConsented] << endl;
}

```

Der `WriteConsensus` liefert ebenfalls den Consented Buffer Index. Darüber hinaus ist die Information wichtig, in welchen Buffer zuletzt geschrieben wurde. Diese steht in der globalen Variablen `s_nLastWritten` zur Verfügung.

Aus dem Consented und dem Last Written Buffer Index wird bestimmt, welches der Safe Buffer ist.

Sind die Daten neu geschrieben worden, so muss dies mittels der Zuweisung `s_nLastWritten = nSafe` bestätigt werden. Nach der Bestätigung, darf auf keinen Fall weiter in diesen Buffer geschrieben werden.

```

void testWriter(int i_nData)
{
    static const int nPermutator[3][3]
        = {{1,2,1}
          ,{2,2,0}
          ,{1,0,0}};

    int nConsented = getWriteConsensus();
    int nSafe      = nPermutator[nConsented][s_nLastWritten];
    s_aBuffer[nSafe] = i_nData;
    s_nLastWritten = nSafe;
}

```

## Initialisierung

Da es möglich ist, dass der Reader bereits vor dem ersten Write ausgeführt werden könnte, muss ein Buffer bereits schon mit einem Defaultwert initialisiert werden (L03). Wir wählen den Buffer mit dem Index 0 als Defaultbuffer. Natürlich muss nun auch die Variable `s_nLastWritten` entsprechend gesetzt werden (L06).

```
L01 void init(int i_nDefault)
L02 {
L03     s_aBuffer[0] = i_nDefault;
L04     s_bSync      = false;
L05     s_bReadSignal = true;
L06     s_nLastWritten = 0;
L07 }
```

Entsprechend müssen auch sämtliche Flags gesetzt werden. Das Sync Flag wird false gesetzt, da noch keine Synchronisation stattgefunden hat. Das Read Signal hingegen muss auf true gesetzt werden, da der WriteConsensus einen neuen Buffer beschreiben soll.

## Test

Folgender Code dient zum sequentiellen Testen<sup>13</sup>:

```
void test()
{
    init(99);
    testReader();
    testWriter(11);
    testReader();
    testReader();
    testWriter(22);
    testReader();
    testWriter(33);
    testReader();
    testWriter(44);
    testReader();
}
```

Es wird folgender Output erwartet:

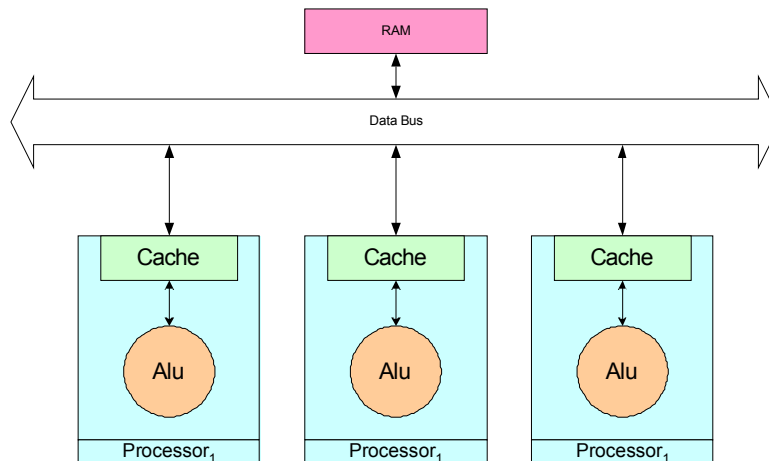
```
Read[0]: 99
Read[1]: 11
Read[1]: 11
Read[2]: 22
Read[0]: 33
Read[1]: 44
```

<sup>13</sup> Obwohl eigentlich zwei Threads am Test beteiligt sein müssten, wird nur Single-Threaded getestet, so dass das Resultat vorhersagbar bleibt.

## Multiprozessor Betriebssysteme

### Cache Problematik

Soll die Consensus Methode auf einem Multi-Prozessor-System eingesetzt werden, so muss darauf geachtet werden, dass der Cache zur richtigen Zeit geleert wird. Nur so kann ein korrektes Funktionieren gewährleistet werden.



Wird der TestAndSet Befehl direkt angewendet, so wird der Cache der CPU höchstwahrscheinlich nicht geleert. Aus diesem Grund muss anstelle von TestAndSet ein spezieller Multiprozessor-Befehl eingesetzt werden, welcher nebst dem atomaren TestAndSet-Befehl ein WriteThrough in den gemeinsamen Speicher ermöglicht.

Da sowohl der Read- als auch der WriteConsensus mit TestAndSet abgeschlossen wird, funktioniert der vorliegende Algorithmus auch in einer Multiprozessorumgebung, sofern ein WriteThrough vom Cache garantiert werden kann.

### Multiprozessor Windows auf X86

```
LONG InterlockedExchange (
    LONG volatile* Target,
    LONG Value
);
```

#### Parameters

*Target*

[in, out] Pointer to the value to be exchanged. The function sets this variable to *Value*, and returns its prior value.

*Value*

[in] Value to be exchanged with the value pointed to by *Target*.

#### Return Value

The function returns the initial value pointed to by *Target*.

### Anwendung

#### TestAndSet für Windows auf X86

```
bool testAndSet(volatile long& i_raFlag)
{
    return InterlockedExchange(&i_raFlag, true)?true:false;
}
```

## Multiprocessor VxWorks auf X86

```
BOOL vxTas
(
    void* pAddress
);
```

### Parameters

pAddress

[in, out] Address to test and set.

### Return Value

TRUE if the value had not been set (but is now), or FALSE if the value was set already.

### Anwendung

#### TestAndSet für VxWorks auf X86

```
bool testAndSet(volatile long& i_raFlag)
{
    return !vxTas(&i_raFlag);
}
```

## Verifikation unter Multiprozessorbetriebssystemen

Soll die beschriebene Methode in einem Multiprozessorbetriebssystem verwendet werden, so ist zu beachten, dass Memory  $\leftrightarrow$  Memory Kopien unterbrochen werden können. Dies gilt sowohl für den Reader als auch für den Writer.

Um die Richtigkeit der beschriebenen Methode überprüfen zu können, schlagen wir vor, den Code so zu modifizieren, dass die Zwischenschritte ersichtlich sind. Dies kann über temporäre Variablen erreicht werden, welche in Register angelegt werden.

```
int getReadConsensus()
{
    s_bSync      = false;
    s_bReadSignal = true;

    register int nReaderPref = s_nLastWritten;
    s_nReaderPref = nReaderPref;
    register int nPref;
    if (testAndSet(s_bSync)==false)
    {
```

```
    nPref = s_nReaderPref;
}
else
{
    nPref = s_nWriterPref;
}
return nPref;
}
```

```
int getWriteConsensus()
{
    register int nConsented;
    if (s_bReadSignal)
    {
        s_bReadSignal = false;

        register nWriterPref = s_nLastWritten;
        s_nWriterPref = nWriterPref;
        if (testAndSet(s_bSync)==false)
        {
            nConsented = s_nWriterPref;
        }
        else
        {
            nConsented = s_nReaderPref;
        }
        s_nConsented = nConsented;
    }
    else
    {
        nConsented = s_nConsented;
    }
    return nConsented;
}
```

# C++ Implementation

## Einführung

Die gezeigte Implementation ist zwar relativ einfach zu verstehen, hat aber alle Nachteile, die globale Variablen mit sich bringen. Trotzdem wurde diese Variante aus didaktischen Gründen gewählt.

Im Folgenden soll eine C++ Lösung entwickelt werden, die beliebig viele Read- und Write-Consensus Paare ermöglicht.

## Shared Data

Als ersten wichtigen Schritt kapseln wir die globalen Daten in einem Objekt. Da der Typ des Buffers je nach Anwendung variiert, ist es naheliegend, diesen separat von den Daten zu behandeln, die von der Algorithmik benötigt werden.

```
L01 class SharedData
L02 {
L03     public:
L04         SharedData()
L05             : m_bSync          (false)
L06             , m_bReadSignal    (true)
L07             , m_nLastWritten   (0)
L08             , m_nReaderPref    (0)
L09             , m_nWriterPref    (0)
L10             {}
L11
L12         volatile long m_bSync;
L13         volatile bool m_bReadSignal;
L14         volatile int  m_nLastWritten;
L15         volatile int  m_nReaderPref;
L16         volatile int  m_nWriterPref;
L17     };
```

Interessant bei dieser Lösung: Gegenüber der Variante mit den globalen Variablen ist hier die elegante Initialisierung im Konstruktor möglich. Auf Zeile L07 wird der Buffer #0 als Default-Buffer bestimmt.

## Buffer

Da der Datentyp je nach Anwendung variiert, wird der Buffer mit Hilfe eines C++ Templates realisiert:

```

L01 template <class T>
L02 class Buffer
L03 {
L04     public:
L05         Buffer(const T& i_raDefaultValue)
L06         {
L07             m_aBuffer[0] = i_raDefaultValue;
L08         }
L09
L10         volatile T& operator[](int i_nIdx)
L11         {
L12             return m_aBuffer[i_nIdx];
L13         }
L14
L15     private:
L16         volatile T m_aBuffer[3];
L17 };

```

Auch hier ist es wichtig, den Buffer als `volatile` zu deklarieren, da mehrere Threads darauf zugreifen.

## ReadConsensus

Sowohl der Read- als auch der WriteConsensus sind nicht vom Buffer abhängig sondern liefern lediglich den korrekten Index, um diesen anzusprechen.

Beide Konsensparteien benötigen Zugriff auf die zur Konsensfindung nötigen gemeinsamen Daten (SharedData). Deshalb wird eine entsprechende Referenz als Membervariable angelegt (L25):

```

L01 class ReadConsensus
L02 {
L03     public:
L04         ReadConsensus(SharedData& i_raSharedData)
L05         : m_raSharedData (i_raSharedData)
L06         {}
L07
L08         int getConsensus()
L09         {
L10             m_raSharedData.m_bSync          = false;
L11             m_raSharedData.m_bReadSignal = true;
L12
L13             m_raSharedData.m_nReaderPref = m_raSharedData.m_nLastWritten;
L14             if (testAndSet(m_raSharedData.m_bSync)==false)
L15             {
L16                 return m_raSharedData.m_nReaderPref;
L17             }
L18             else
L19             {
L20                 return m_raSharedData.m_nWriterPref;
L21             }
L22         }
L23
L24     private:
L25         SharedData& m_raSharedData;
L26 };

```

## WriteConsensus

Der WriteConsensus Code ist ähnlich einfach. Zusätzlich stehen neu aber noch die zwei Methoden getLastWritten (L28) und setLastWritten (L33) zur Verfügung, damit der Writer einfacher implementiert werden kann.

```

L01 class WriteConsensus
L02 {
L03     public:
L04         WriteConsensus(SharedData& i_raSharedData)
L05             : m_raSharedData (i_raSharedData)
L06               , m_nConsented (0)
L07             {}
L08
L09         int getConsensus()
L10         {
L11             if (m_raSharedData.m_bReadSignal)
L12             {
L13                 m_raSharedData.m_bReadSignal = false;
L14
L15                 m_raSharedData.m_nWriterPref = m_raSharedData.m_nLastWritten;
L16                 if (testAndSet(m_raSharedData.m_bSync)==false)
L17                 {
L18                     m_nConsented = m_raSharedData.m_nWriterPref;
L19                 }
L20             }
L21             else
L22             {
L23                 m_nConsented = m_raSharedData.m_nReaderPref;
L24             }
L25             return m_nConsented;
L26         }
L27
L28         int getLastWritten() const
L29         {
L30             return m_raSharedData.m_nLastWritten;
L31         }
L32
L33         void setLastWritten(int i_nLastWritten)
L34         {
L35             m_raSharedData.m_nLastWritten = i_nLastWritten;
L36         }
L37
L38     private:
L39         SharedData& m_raSharedData;
L40         int         m_nConsented;
L41 };

```

## Anwendung

Der Writer lässt sich nun genauso kompakt formulieren wie in der C-basierten Lösung. Zusätzlich müssen wir nun aber sowohl dem Reader als auch dem Writer jeweils die Referenz auf den entsprechenden Consensus und den Buffer mitgeben:

```

L01 void testWriter(WriteConsensus& i_raWriteConsensus,
L02                 Buffer<int>&    i_raBuffer,
L03                 int            i_nData)
L04 {
L05     static const int nPermutator[3][3] = {{1,2,1}
L06                                           ,{2,2,0}
L07                                           ,{1,0,0}};
L08
L09     int nLastEngaged    = i_raWriteConsensus.getConsensus();
L10     int nLastWritten   = i_raWriteConsensus.getLastWritten();
L11     int nWriteIdx      = nPermutator[nLastEngaged][nLastWritten];
L12     i_raBuffer[nWriteIdx] = i_nData;
L13     i_raWriteConsensus.setLastWritten(nWriteIdx);
L14 }

```

Der Reader ist gewohnt einfach:

```

L01 void testReader(ReadConsensus& i_raReadConsensus,
L02                Buffer<int>&    i_raBuffer)
L03 {
L04     int nEngaged = i_raReadConsensus.getConsensus();
L05     cout << "Read[" << nEngaged << "]: " << i_raBuffer[nEngaged] << endl;
L06 }

```

Die Anwendung gestaltet sich dank der Objekt-Orientierten Syntax intuitiv. Zusätzlich ist es nun möglich, beliebig viele Instanzen der Read- / WriteConsensus-Paare aufzusetzen.

```

L01 void test()
L02 {
L03     Buffer<int>    aBuffer(99);
L04
L05     SharedData    aSharedData;
L06     ReadConsensus aReader(aSharedData);
L07     WriteConsensus aWriter(aSharedData);
L08
L09     testReader(aReader, aBuffer);
L10     testWriter(aWriter, aBuffer, 11);
L11     testReader(aReader, aBuffer);
L12     testReader(aReader, aBuffer);
L13     testWriter(aWriter, aBuffer, 22);
L14     testReader(aReader, aBuffer);
L15     testWriter(aWriter, aBuffer, 33);
L16     testReader(aReader, aBuffer);
L17     testWriter(aWriter, aBuffer, 44);
L18     testReader(aReader, aBuffer);
L19 }

```

## Schlussüberlegungen

---

Dank des Consensus Konzeptes wird es möglich, ein asynchrones, loop- und wait-freies Update zu erreichen. Die Daten werden dabei über drei Buffer ausgetauscht.

Dank des Einsatzes des hardwarenahen Befehls TestAndSet kann der Datenaustausch mit maximaler Geschwindigkeit erfolgen.

Da die Buffer schon beim Startup des Systems angelegt werden können, eignet sich das System auch für kritische Echtzeitaufgaben.

Der Consensus-gesteuerte Datenaustausch kann immer dann eingesetzt werden, wenn der Reader nicht darauf angewiesen ist, alle Datenupdates zu behandeln und auch mit einem älteren Datenstand arbeiten kann.

Wir sind überzeugt, dass der beschriebene Algorithmus in vielen Systemen erfolgreich eingesetzt werden kann.

## Literatur

---

- [1] Gamma, Erich et al.: Design Patterns.  
Addison Wesley, 1995. ISBN 0-201-63361-2.
- [2] Stroustrup, Bjarne: The C++ Programming Language, Third Edition.  
Addison Wesley, 1997. ISBN 0-201-88954-4.
- [3] Lippmann, Stanley B., Lajoie, Josée: C++ Primer, Third Edition.  
Addison Wesley, 1998. ISBN 0-201-82470-1.
- [4] Lippmann, Stanley B: Inside the C++ Object Model.  
Addison Wesley, 1996. ISBN 0-201-83454-5.
- [5] Coplien, James O.: Advanced C++ Programming Styles and Idioms.  
Addison Wesley, 1992. ISBN 0-201-54855-0.
- [6] Meyers, Scott: Effective C++, Second Edition.  
Addison Wesley, 1998. ISBN 0-201-92488-9.
- [7] Meyers, Scott: More Effective C++.  
Addison Wesley, 1998. ISBN 0-201-63371-X.
- [8] Stroustrup, Bjarne: The Annotated C++ Reference Manual.  
Addison Wesley, 1995. ISBN 0-201-51459-1.
- [9] Chen, Jing; Burns, Alan: Asynchronous Data Sharing in Multiprocessor Real-Time Systems Using Process Consensus.  
<http://citeseer.nj.nec.com/114960.html>.