

BI3 Specification

## Secure isolated processes

# BI3-TBE Emulator

The BI3-TBE resolves the Criteria of Abstraction defined by TAIS and follows the general format presented in the BI3-TAE:

- **Circuit layout**
- **Primitives**
- **Factory load**
- **Booting**
- **Normal operation**
- **Shutdown**
- **Interfaces**
- **Special**

Note that the TBE enables simple networking by way of passing a newly introduced primitive called the *Sealed-Duplex* to the TBE process at boot time. As a result, the TBE process can also be referred to as an *Agent*

because it resembles a mobile agent. Mobile agents are able to move from machine to machine virally without subverting the architecture; this capability is acquired in the TAIS Gamma Environment (TGE).

## Circuit layout

The BI3-TBE Emulator inherits the BI3-TAE Emulator but replaces the Fixed-TAE-Process-Images by way of loading Agents with the Execute primitive, whose images are stored in the Slow-Memory microprocessor.

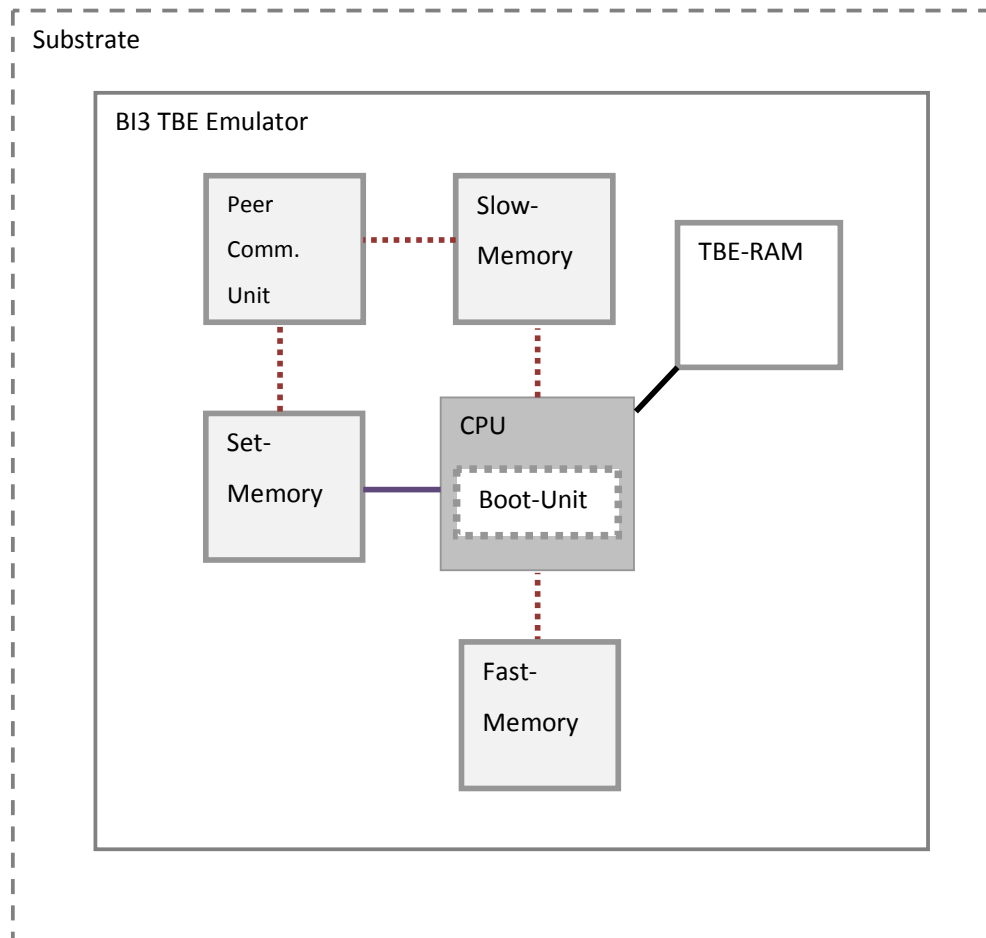
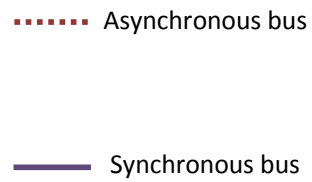


Figure 1 – The BI3-TBE Emulator is network capable and useful when compared to the BI3-TAE

Resolution of criteria of abstraction for the BI3-TAE Emulator				
Factors		Design-Time Factors		
TBE Factor <sup>1</sup>	Factor summary	Resolved in TAIS	Resolved in BI3-TBE	Nature of resolution
1@B1	TBE-RAM replaces RAE-RAM	No	No	N/A
2@B1	TBE process replaces TAE process	No	No	N/A
3@B1	TBE code segment loaded from Slow-Memory	Yes	N/A	Slow-Memory stores TBE process code segment
1@B2a	TBE process accesses Boot-Parameter	No	No	N/A
2@B2a	Boot-Parameter determined by implementation	No	No	N/A
3@B2a	Boot-Parameter has TBE process Execute primitive assigned	Yes	N/A	Structural definition
1@B2b-1	Each SOWA is keyed	No	No	N/A
2@B2b-1	At least one SOWA present	No	Yes	Slow-Memory and Fast-Memory
1@B2b-2	Can modify Set	No	Yes	Set methods specified
2@B2b-2	Introspective scope	No	No	N/A
3@B2b-2	May not reference Sets on other Agents	No	No	N/A
4@B2b-2	Consistent ordering	No	No	N/A
5@B2b-2	Execute primitive assigned to Set	Yes	N/A	Structural definition
6@B2b-2	1 SOWA for each type assigned to Set	Yes	N/A	Structural definition
7@B2b-2	Resolution of persistence	No	Yes	Slow-Memory (Yes) and Fast-Memory (No)
8@B2b-2	Permanent Whole and Float assigned to Set	Yes	N/A	Structural definition

<sup>1</sup> Reference “The 7 Steps of Absolute Information Security with TAIS, version 0.915”

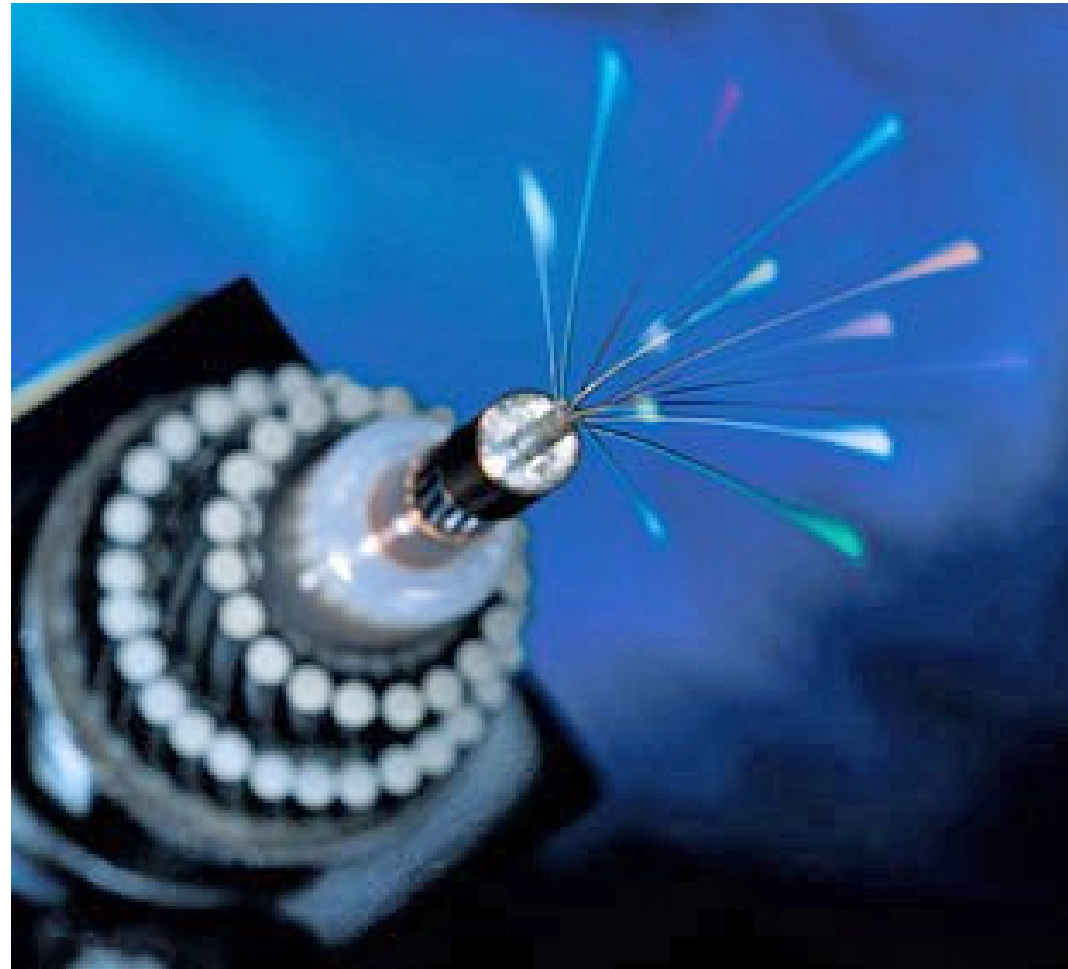
9@B2b-2	1 Sealed-Duplex assigned to Set	Yes	N/A	Structural definition
1@B2b-3	TBE process accesses Float, Whole and Binary	No	No	N/A
2@B2b-3	Encoding undefined	No	No	N/A
1@B2b-4	Sealed-Duplex is viral	Yes	N/A	Structural definition
2@B2b-4	Sealed-Duplex can merge	Yes	N/A	Structural definition
3@B2b-4	Sealed-Duplex can produce Unsealed-Duplex	Yes	N/A	Structural definition
4@B2b-4	Unsealed-Duplex operates when partner unsealed too	Yes	N/A	Structural definition
5@B2b-4	Network determines data integrity	No	No	N/A
6@B2b-4	Encoding undefined	No	No	N/A
7@B2b-4	Unsealed-Duplex transmits other primitives	Yes	N/A	Structural definition
8@B2b-4	New Unsealed-Duplex shares parent bandwidth	Yes	N/A	Structural definition
1@B3a	All Set references copied	Yes	N/A	Structural definition
1@B3b	Communication via a minimum number of Context-Handlers	No	No	N/A
2@B3b	Context-Handlers are scope limited	No	No	N/A
3@B3b	Context-Handlers contain no errors	No	No	N/A
4@B3b	Non-trusted code must reside on its own TBE process	No	No	N/A
5@B3b	Non-trusted code divided into maximum modules	No	No	N/A

## Primitives

The BI3-TBE introduces a number of primitives that are reflected in TAIS:

- Execution
- Float
- Whole
- SOWA
- Sealed-Duplex
- Unsealed-Duplex
- Set

Primitive are used inside code executing in the CPU to reference other memory types or to pass parameters to these memory types. In the BI3-TBE the basic TAIS primitives are mapped to the BI3 and new related primitives are introduced to handle parameters and extend the original behaviour.



<b>BI3 TBE Primitive Translation</b>				
<b>TAIS abstract primitive</b>	<b>Resolved BI3-TBE primitive</b>	<b>Referenced primitives</b>	<b>Referenced microprocessor</b>	<b>Description</b>
Execution	Execution	Set	Slow-Memory & Set-Memory	Software instructions are stored in Slow-Memory
Float	Float	Duplex & Set	TBE-RAM & Set-Memory	Floating-point number
	Float [ ]	N/A	TBE-RAM	Array of Floating-point numbers
Whole	Whole	Unsealed-Duplex & Set	TBE-RAM & Set-Memory	Integer number
	Whole [ ]	Unsealed-Duplex & Set	TBE-RAM	Array of Integer numbers
Binary	Binary	N/A	TBE-RAM	Bitwise management of a data block
	Binary[]	N/A	TBE-RAM	Array of Binary blocks
SOWA	Slow-Memory	Unsealed-Duplex & Set	Slow-Memory & Set-Memory	SOWA type on a low-performance bus
	Fast-Memory	Unsealed-Duplex & Set	Fast-Memory & Set-Memory	SOWA type on a high-performance bus
N/A	Null	Unsealed-Duplex & Set	TBE-RAM, Slow-Memory & Set-Memory	Conventional null
N/A	Boolean	N/A	TBE-RAM	Conventional boolean
Sealed-Duplex	Flow-Junction	N/A	TBE-RAM	Partial Duplex link
Unsealed Duplex	Flow	N/A	Fast-Memory & Slow-Memory	Chained Sealed-Duplex links that are bound on both ends to TBE processes and have been unsealed
Set	Set	Duplex	Set-Memory	Computational representation of mathematic Set structure
	Trilarity	Set	Set-Memory	Special parametric type for Set manipulation
	Polarity	Set	Set-Memory	Special parametric type for Set manipulation
	Lockmode	Set	Set-Memory	Special parametric type for Set manipulation
	Monitor	Set	Set-Memory	Establish events based on Set conditional triggers

**Execution primitive**

The Execution primitive represents the code segment and is stored in the Slow-Memory area. Java implementations (such as the Bluebrick™ Emulator) should take care to not load the same class from multiple Agents. If two Agents use the same Slow-Memory

reference for representing code, then they may be utilising the same class within the implementation. The problem is that a shared class would provide a covert channel via static variables and inheritance of objects that share public static variables. To avoid this, the

emulator should create each class with unique names internally so that there is no referential connection between them. *ClassX* that is used 3 times should be renamed to *ClassX1*, *ClassX2* and *ClassX3* or an equivalent internal representation.

**Slow-Memory primitive**

Slow-Memory is a TAIS SOWA primitive type that is optimised for long-term storage of massive files and will most likely be ultimately stored on Hard-drives in the implementation(as at 2011). A Slow-Memory is only stored permanently when it is attached to a Set and that Set is referenced by at least one other Set already present in the Set-Memory. For any Set to be accessible by the TBE process after restart, it needs to be

located somewhere in the Boot-Parameter that was passed to the TBE process on boot.

```

public class SlowMemory2()
{
    public      SlowMemory3() ;

    public      FlowJunction slowMemoryAsFlowJunction buildSlowMemory4    () ;

    public      boolean      isSame                  compareSlowMemory    (SlowMemory slowMemoryToCompare) ;

    public      whole        slowMemorySize         getSize() ;

    public      whole        hashCode               getHashCode5() ;

    public      whole        status6              getSlowMemoryStatus() ;

    public      FlowJunction returnedFlowJunction7 retrieveSlowMemory() ;

}

```

---

<sup>2</sup> All Slow-Memory is persistent when attached to a Set after an elegant shutdown and restart.

<sup>3</sup> Creates a new Slow-Memory for which the content is created from the FlowJunction received by a call to **buildSlowMemory()** and the Slow-Memory is completed when a `closeForWriting()` is called on the resulting Flow. Note, you must always call `closeForWriting()` on the FlowJunction or else the thread will block when reading the hash-code, size or other functions that require a completed Slow-Memory in order to continue to the next instruction.

<sup>4</sup> Returns a Flow-Junction whose Flow can only be written to in order to create the Slow-Memory content. Any read instruction will call a `FlowClosedForReadingException()` of the Flow class. Any channel (float, whole, Slow-Memory or Fast-Memory) except Set can be used to generate the resulting Slow-Memory.

<sup>5</sup> Returns a hashcode whose format is specific to the implementation and will block until the Slow-Memory has been transferred.

<sup>6</sup>

- 1 = Fully transferred or created and has a permanently completed image
- 2 = The image is still being created
- 3 = The image is partially completed and has failed but the system is currently attempting to recover the data
- 4 = The image is partially completed, the system has abandoned any recovery attempt

<sup>7</sup> This Flow-Junction produces a special type of Flow where the Slow-Memory is read with consecutive calls to any of the Flow read functions except `readSet()` such as `Flow.readWhole()` and `Flow.readFloat()`. A call to `readSlowMemory()` or `readFastMemory()` will retrieve the remainder of the file. Any write calls will throw a `FlowClosedForWritingException()`.

**Fast-Memory primitive**

Fast-Memory is also a TAIS SOWA primitive type that is optimised for short-term manipulation and fast access of small memory blocks such as video or images on screen and

will most likely be realised in memory in the implementation(as at 2011). Being a SOWA type, Fast-Memory is therefore identical to Slow-Memory, except Fast-Memory is stored

fully on the BI3-TBE Emulator, so read calls will not block. Fast-Memory lacks the stream based operations of Slow-Memory.

```
public class FastMemory8()
{
    public FastMemory ( binary[] binaries) ;
    public FastMemory( whole [] wholes) ;
    public FastMemory( float [] floats) ;
    public FastMemory9( Flow flow, whole numberOfBlocks) throws memoryShortageException(SlowMemory fastMemoryImage) ;
    public boolean      isEqualTo          compareFastMemory(FastMemory fastMemory) ;
}
```

---

<sup>8</sup> All `FastMemory` primitives are lost after restart, so `SlowMemory` primitives must be used for permanent storage instead.

<sup>9</sup> Creates a `FastMemory` from a resulting `Flow` by calling `readWhole()` of the `Flow` class. The constructor will block only if the `Flow` has no more written data in its buffer and the `numberOfBlocks` read has not yet been reached.

```

public float      float      getFloatAt(whole index) ;
public whole      size       getSize() ;
public whole      whole      getWholeAt(whole index) ;
public binary[]   binaryData[]10 retrieveMemory() ;
public float []   wholeArray11 retrieveFloatMemory() ;
public whole []   wholeArray12 retrieveWholeMemory() ;
}

```

---

<sup>10</sup> Creates a single binary array.

<sup>11</sup> Returns `null` if no FastMemory has been allocated to this set

<sup>12</sup> Returns `null` if no Fast Memory has been allocated to this set

## Flow-Junction

A *Flow-Junction* is an implementation of the TAIS Unsealed-Duplex. It is a bidirectional stream of data and Sets that is presented so that data can only be read and written from its end-points (see *Flow* next). The Flow-Junction is useful because it can connect to other Flow-Junctions to form a chain that can link Agents (TBE processes) within the same Node (BI3-TBE Emulator) and across any number of other Nodes. Flow-Junctions form a switched network that overlays the inherent peer-to-peer connectivity of the Nodes themselves in much the same way that the original switched telephone network operates. Switched networks contrast to TCP/IP packet delivery and its associated jitter

that is often experienced with VOIP calls. Once a path of Flow-Junctions has been determined, it is locked down and a Flow is generated. The switched data path ensures that variability in latency (jitter) is zero and a reliable fixed bandwidth connection is delivered between end-points.

Such an arrangement is designed to address a major weakness of TCP/IP, which has a more chaotic architecture where packets are forwarded beyond the control of the application using it. Flow-Junctions on the other hand give the application complete control over which BI3-TBE Emulators that are used to transmit the data for the Flow-Junction. Although the security benefits are

managed by the TAIS Delta Environment (TDE), the ability to control the data path allows applications to build redundancy into their transmission, reduce risk of failure and even eliminate stream disruption in the event that one of the redundant Flow-Junction paths failed.

When the Flow-Junction is fully connected, information can be retrieved from each end-point by producing a *Flow* object with the `unsealFlow()` method. Once the Flow-Junction has had the `unsealFlow()` method called, it can no longer be joined to other Flow-Junctions by using the `connectFlow()` method.

```

public class FlowJunction()
{
    public boolean success13          connectSealedFlowJunction14 (    FlowJunction sealedFlowJunction) ;
    public boolean connectionStatus    isConnectedToFlowJunction15() ;
    public boolean sealedStatus        isSealed16() ;
    public Flow flow17                unsealFlow() ;
}

```

---

<sup>13</sup> Value is false if Flow could not be connected because it is already unsealed or closed

<sup>14</sup> The function connects two Flows that are still sealed

<sup>15</sup> Returns true if this FlowJunction is connected to another FlowJunction regardless of whether it is sealed or unsealed at the terminating end

<sup>16</sup> Returns true if the FlowJunction remains sealed or false if the *unsealFlow()* method has been called

<sup>17</sup> Value is null if the Flow has already been unsealed

## Flow

A *Flow* is a readable object that can be extracted from a Flow-Junction with the `unsealFlow()` method. The life of a Flow ends when the stream is closed by both ends and the last bit and set is read, which results in a `FlowClosedException()`.

Flows have 6 independent asynchronous sub channels inside them for Float, Whole, Set, Fast-memory, Slow-Memory and Flow-Delayed Write

If a connection between a partner is broken, then an `FlowAbruptClosedException` is thrown. However, some of the data that may be written will be buffered on the localhost and this may not be written. It is essential for Agents to manage their own data error correction.

Junction. In other words, data will arrive in the same order that it was written for each channel. For example, the `readFloat()` method will block until the `writeFloat()` method is called on the opposite end of the Flow.

All Fast-Memory moving down a Flow must be regenerated on the recipient Agent. Agents have a limited resource allocation and if the

Fast-Memory is too large then a `memoryShortageException()` is thrown. Since Fast-Memory is transient, it must be converted to Slow-Memory and stored on a Set in order for it to permanently remain in the Agent data store.

Availability

Transmission of data along the Flow occurs on separate primitive channels. Accessibility is immediate, so that when reading any channel, the start of the Fast-Memory or Slow-Memory stream must be made available immediately without waiting for the entire file to be transmitted first. Calls to `readSlowMemory()` or `readFastMemory()` will not block providing there is at least one block of data in the buffer. The same is true for transmission

of Sets for which large Slow-Memory or Fast-Memory data is attached – they are immediately available. However, Sets are atomic and are not able to be read until they are fully transmitted.

To access the actual data, the `retrieveSlowMemory()` or `retrieveFastMemory()` method calls are made that both return a Flow-Junction. The Flow-Junction interface allows data to be

accessed serially as soon as it becomes available (such as in a bidirectional voice call).

The data can only be accessed randomly in a Fast-Memory with the `retrieveMemory()` call, which returns the array containing the entire sequence of N-Bit blocks. Note that this method will block until the entire file becomes available.

```

public class Flow() throws FlowAbruptClosedException, FlowClosedForWritingException18(), FlowClosedForReadingException19()
{
    public      void      abruptClose20(Set reason) ;
    public      void      closeForWriting21() ;
    public      boolean    isNaturallyClosed      isNaturallyClosed22() ;
}

```

---

<sup>18</sup> Called if an Agent calls `closeForWriting()` and then tries to write something.

<sup>19</sup> Called when an Agent is attempting to read excessively after its peer has called `closeForWriting()`. This exception is thrown when the last data chunk for any single channel (`float`, `whole`, `Set`, `Execute`, `FastMemory` or `SlowMemory`) has already been read and the thread attempts to read again so that without the exception it would otherwise block forever.

<sup>20</sup> The `abruptClose()` method may be called by either end of the channel to terminate the session and release the resource without waiting for the remaining data in the buffer to be read. If any data is read or written by either end of the `Flow`, then a `FlowAbruptedClosedException()` is thrown. This method is automatically called by the BI3-TBE Emulator if the physical or logical connection is broken.

<sup>21</sup> Naturally closes a `Flow`. When both ends of a `Flow` are closed for writing and all the data read, then it throws a `FlowNaturalClosedException()` only if more data is read or written to `Flow`.

<sup>22</sup> The `Flow` is considered naturally closed when both endpoints have called `closeForWriting()` and each of the `float`, `whole` and `Set` streams have been fully read by both endpoints. Note that a call to `isNaturallyClosed()` can yield `true` without a `NaturalCloseException()` being thrown because the latter only occurs if an additional read is made after `isNaturallyClosed()` yields `true`.

```

public      FastMemory  fastMemory23,
            SlowMemory  fastMemoryImage24      readFastMemory25() ;

public      FastMemory  fastMemory,
            SlowMemory  fastMemoryImage,
            whole        numberOfBytesRead,
            boolean     doesReadTerminateSection26 readFastMemory27(whole limit) ;

public      float      float      readFloat () ;

public      FlowJunction flowJunction28      readFlowJunction29() ;

public      SlowMemory slowMemory      readSlowMemory30() ;

```

---

<sup>23</sup> Contains the read `FastMemory` or if the image is too big, it will contain `null` and the following parameter `fastMemoryImage` will contain the read `FastMemory` instead.

<sup>24</sup> Contains the `FastMemory` image if the previous parameter `FastMemory` is null when the image is too large.

<sup>25</sup> Will read the next `FastMemory`, provided that it has been written on the other end of the `Flow`, else it will block.

<sup>26</sup> Returns true if this ends a series of partial reads of a block of `FastMemory`

<sup>27</sup> Will read the next `FastMemory` up to a maximum number of bytes (`limit`), provided that it has been written on the other end of the `Flow`, else it will block.

<sup>28</sup> Returns null if this `FlowJunction` has had the `connectSealedFlowJunction()` call made previously.

<sup>29</sup> Obtains a new `FlowJunction` embedded inside another that will share the parent `FlowJunction` bandwidth capacity.

<sup>30</sup> Reads the next `SlowMemory`, providing that next one has been written on the other end of the `Flow`.

```

public      SlowMemory slowMemory,
           whole      numberOfBlocksRead,

boolean     doesReadTerminateSection31      readSlowMemory32(whole limit) ;

public     whole whole      readWhole () ;

public Set  set,
           SlowMemory fastMemoryImage33      readSet () ;

public Set  set,
           SlowMemory fastMemoryImage34      readSet      (TransmissionObserver transmissionObserver35) ;

public     void      writeFastMemory36      (FastMemory fastMemory) ;

public     void      writeFloat      (float float) ;

public     void      writeFlowJunction37 (FlowJunction flowJunction) ;

```

---

<sup>31</sup> Returns true if this series of partial reads encounters the end of the stream

<sup>32</sup> Will read the next Slow-Memory up to a maximum number of bytes (*limit*), provided that it has been written on the other end of the Flow, else it will block.

<sup>33</sup> Normally *fastMemoryImage* is null unless *status* = 3 when it contains the *FastMemory* image of *set*.

<sup>34</sup> Normally *fastMemoryImage* is null unless *status* = 3 when it contains the *FastMemory* image of *set*.

<sup>35</sup> Enables the variable length primitives to be monitored if *writeSet* () uses the non-blocking mode.

<sup>36</sup> Function *writeFastMemory* () will write the next *FastMemory* to the Flow, but blocks if the previous *FastMemory* has not been read by the other Flow end.

<sup>37</sup> Call is ignored if this *FlowJunction* has had the *connectSealedFlowJunction* () call made previously

```

public void writeWhole (whole whole) ;
public void writeSlowMemory38 (SlowMemory slowMemory) ;
public void writeSet39 (Set set) ;
public void writeSet40 (Set set,
                        TransmissionObserver transmissionObserver) ;
}

public abstract class TransmissionObserver
{
    public whole getNumberOf64BitBlocksWritten() ;

    public abstract slowMemoryFailed() ;

    public abstract slowMemorySuccessfullyTransmitted() ;
}

```

---

<sup>38</sup> Function writeSlowMemory() will block until closeForWriting() is called on the previous SlowMemory that was written with writeSlowMemory(), unless that SlowMemory was created with the array constructor so it was available in its entirety.

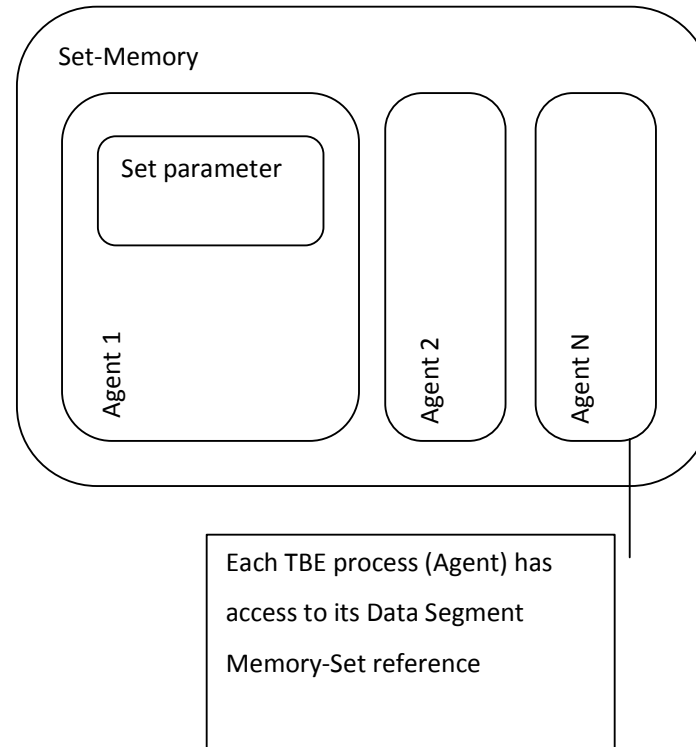
<sup>39</sup> Writes a Set across the Flow so that the opposite readSet() will automatically block until all the variable length primitives (SlowMemory, FastMemory and Execute are completely written).

<sup>40</sup> Writes a Set across the Flow like the writeSet(Set set) method above, except that the opposite readSet() will not block. The TransmissionObserver object is used to monitor if the set is completed or fails.

**Set-Memory**

Set-Memory will boot to its last saved image that occurs on shutdown. An abrupt failure will result in all the Set-Memory being lost up to the point of the last shutdown<sup>41</sup>. However, individual Agent Boot-Parameter Set and all its associated data structures are stored separately whenever the `saveAgent()` method is called in the `Agent` class.

Set-Memory is extremely basic in the BI3-TBE Emulator and matures further in the BI3-TGE Emulator.



<sup>41</sup> Some implementation such as Bluebrick™ may choose to recreate the BI3-TBE Emulator memory structure (addressed more closely in the BI3-TGE) when run in high-reliability mode. In the event that a sudden shutdown occurs and the Agent has not called `saveAgent()`, the BI3-TBE Emulator can take extended booting time to regenerate the Set-Memory state to a point prior to the crash.

```

public class Set()
{
    public Set(whole type) ;

    public Set42() ;

    public void                addMonitor(Monitor monitor) ;

    public void                deleteChild43() ;

    public void                deleteAllChildren() ;

    public Set      retrievedChild44 getChild() ;

    public trilarity cursorShift      getDeleteBehaviour45() ;

    public FastMemory fastMemory      getFastMemory() ;

    public float      floatValue      getFloat() ;

    public FlowJunction flowJunction getFlowJunction46() ;

    public whole      numberOfChildren getNumberOfChildren() ;

    public whole      position        getPosition47() ;
}

```

---

<sup>42</sup> Creates a default Set of Type = -1

<sup>43</sup> Deletes the current child with the default behaviour of the Set.

<sup>44</sup> Is null if there is no Child

<sup>45</sup> Returns the deletion behaviour set with `setDeleteBehaviour()`.

<sup>46</sup> Retrieves the Flow-Junction that has been assigned to this Set.

<sup>47</sup> Where the first position is represented by 1 (unlike traditional computer offsets where 1 = 0)

```

public SlowMemory  slowMemory      getSlowMemory() ;

public whole      setType          getType() ;

public whole      wholeValue      getWhole() ;

public void

insertSet(      Set childSet,

                trilarity trilarity,

                polarity polarity) ;

public Set        newChildSet      insertNewSet(whole type,

                trilarity trilarity,

                lockmode lockmode) ;

public void

setFastMemory(FastMemory fastMemory) ;

public void

setFloat(      float float) ;

public void

setFlowJunction48(FlowJunction flowJunction) ;

public void

setSlowMemory49(SlowMemory slowMemory) ;

public void

setWhole(      whole whole) ;

public void

shiftBookmark(      polarity polarity) ;

```

---

<sup>48</sup> Stores a Flow-Junction onto a set. The Flow-Junction assignment is transient and will be wiped on restart.

<sup>49</sup> Assigns the `SlowMemory` parameter to the Set but it will block until the `SlowMemory` parameter that was created with a Flow-Junction and the resulting Flow has the `closeForWriting()` method called. This is an important function because all Slow-Memory created in memory will not be stored until this method is called. As a result, the Set-Memory can be said to be complete, as no partial Slow-Memory writes are able to be stored inside it. In this way, implementations of the BI3 can avoid storing transient Slow-Memories such as during the transmission of voice communication that is not to be logged.

```

public void          shiftBookmark(          polarity polarity,
                                                    whole numberOfShifts) ;

public void          shiftBookmarkToBoundary(  polarity polarity) ;
}

```

### Monitor primitive

Monitors deliver events to changes in the Set structure or primitives associated with Sets.

Like Fast-Memory, Monitors are not saved as

part of the Set structure and need to be created dynamically on each run by the Agent.

```

public class Monitor
{
    //Base identification - ignored during implementation
}

```

```
public abstract class InsertChildMonitor inherits Monitor
{
    public abstract void eventTriggered50 (    Set parent,
                                             Set insertedChild,
                                             trilarity trilarity,
                                             lockmode lockmode) ;
}
```

```
public abstract class deleteChildMonitor inherits Monitor
{
    public abstract void eventTriggered51 (    Set parent, Set deletedChild) ;
}
```

---

<sup>50</sup> Implemented by Agent to receive the event when a set is inserted into the set being monitored

<sup>51</sup> Implemented by Agent to receive the event when a child is deleted from the Set being monitored

Example of Monitor use:

```
class MyMonitor extends deleteChildMonitor
{
    public void eventTriggered(Set set) ;
    {
        //Handle event here
    }
}

class MyClass
{
    Set set = new Set() ;
    MyMonitor myMonitor = new MyMonitor() ;

    public MyClass()
    {
        Set.addMonitor(myMonitor) ;
    }
}
```

## Factory loading

The Factory loading of the BI3-TBE Emulator is similar to the TAE Emulator in that it is likely to occur the first time that the build is executed.

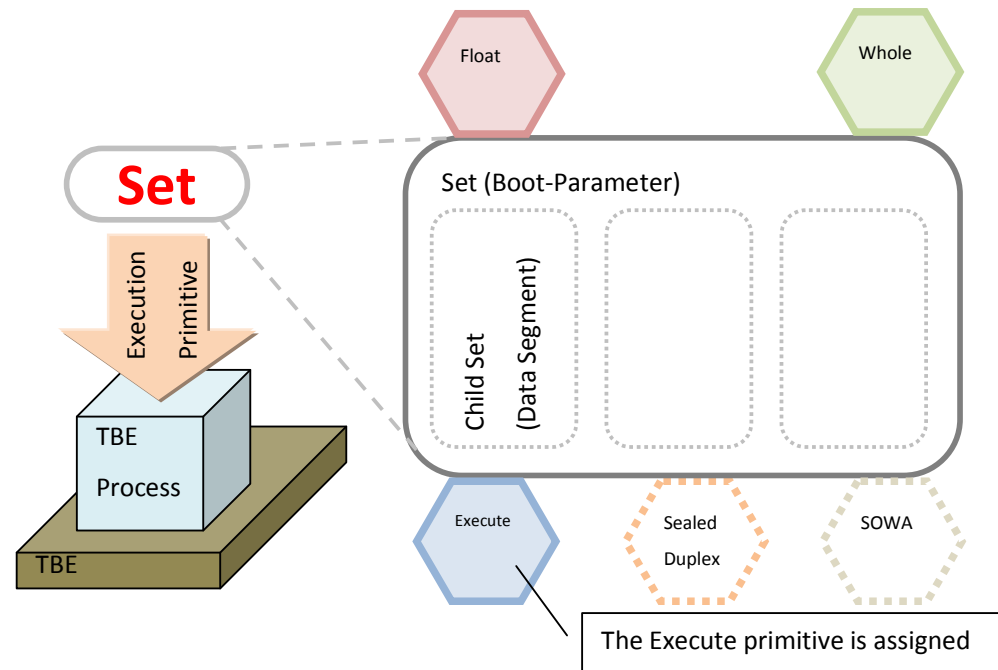
During factory load, the potential TBE process binary images (Agents) must be scanned and stored into the Slow-Memory microprocessor.

A software implementation is likely to provide privileged access to the `factoryLoad()` call so that the actual Agent code can be accessed. Once the Agent code is loaded in, it is scanned to ensure that it contains no dependencies (linked by way of method calls) to any other code. If compliant, the code is converted into some binary format and stored directly in the Slow-Memory microprocessor and tagged as being an Execute primitive. As such, there is no mapped Slow-Memory reference to access this Execute primitive. The only reference to the Execute primitive is assigned to the Set that is passed to the Agent as its boot parameter.

# Booting

The booting of the BI3-TBE Emulator is also similar to the BI3-TAE, but the loading of the Agents is different.

1. All microprocessors are powered.
2. All microprocessors send a signal to the CPU indicating that they function normally and have paused while they await further instruction from the CPU (on their bus) before commencing.
3. The Boot-Unit instructs the CPU to load the code segment from the Slow-Memory microprocessor into the TBE-RAM for each TAE process (Agent).
4. The Boot-Unit constructs<sup>52</sup> the Boot-Parameter for each Agent.
5. The CPU signals to the Boot-Unit when loading is completed
6. Either immediately or on some implementation specific cue, the Boot-Unit signals the CPU to begin normal operation. Prior to this step, the TBE is primed for use.



<sup>52</sup> TAIS Factor Beta 2@2a leaves the contents of this factor entirely to the implementation.

```

public class BI3TBE
{
    public BI3_TBE() ;

    public void                boot() ;

    public void                factoryLoad() ;

    public Set agentList53    getLoadedAgents() ;

    public whole totalNumberOfSlowMemory getTotalNumberOfReferencedSlowMemory() ;

    public whole totalSizeOfSlowMemory getGrossSizeOfSlowMemoryImage() ;

    public boolean      operationStatus54 getOperationStatus() ;

    public void                shutdown() ;
}

```

---

<sup>53</sup> Each child has a Slow-Memory that contains the Agent name, the whole contains the code size in bytes.

<sup>54</sup> Returns true if BI3 is operational or false if it is shutdown.

## Normal operation

Normal operation is identical to the BI3-TAE Emulator in that Agents share the CPU processor.

In addition to the instruction capabilities of the BI3-TAE, the instructions may:

- Change the state of the Slow-Memory, Fast-Memory or Set-Memory microprocessors by utilising the appropriate primitives
- Transmit primitives to another BI3-TBE Node (BI3-TBE Emulator) via a Sealed-Duplex primitive that is supplied within the Boot-Parameter Set.
- The TBE Process may subdivide itself into multiple threads, but each of these threads must share the

allocated process for each Agent. Like the BI3-TAE, when another Agent on the BI3-TBE halts, the other Agents do not gain access to this additional process – this process must go to waste in order to guarantee stability.

The TAIS Gamma Environment (TGE) and the BI3-TGE Emulator address this issue with a sophisticated reservation model.

```

public abstract class Agent() inherits BI3ElementInterface
{
    public abstract55 void bootAgent56( Set bootParameter) ;
    public void killThisAgent57() ;
    public void saveAgent58() ;
    public void selfDestruct59() ;
    public abstract Set newSet60(Whole type) ;
    public abstract FastMemory newFastMemory(Whole []) ;
    public abstract SlowMemory newSlowMemory(Whole []) ;
    public FlowJunction positiveFlowJunction,
        flowJunction negativeFlowJunction createFlowJunctionPair() ;
}

```

---

<sup>55</sup> Implemented by the Mobile Agent inheriting Agent class

<sup>56</sup> Point of entry of TBE process (Agent) at boot

<sup>57</sup> Immediately and permanently removes the Agent making the call (self-kill) from the Node. All threads and data storage are destroyed. All digital debris is eliminated by changing any Slow-Memory store passwords to an unreferenced random code and physically overwriting disk sectors.

<sup>58</sup> Some implementations will automatically log any changes to Sets at runtime, in which case, a call to saveAgent() is ignored as it is redundant.

<sup>59</sup> Destroy this agent

<sup>60</sup> NewSet() is called to create a new Set

## Shutdown

Shutdown of the TBE is similar to the Tae in that it occurs when the implementation triggers the `shutdown()` method of the `BI3TBE` class.