Network Embedded Programming

A programming assignment about SPI, publish/subscribe, directed diffusion and testing.

250478	**Jonas Fonseca Madsen**	`<fonseca@diku.dk>`
140881	**Bo Gotthardt-Petersen**	`<bogp@diku.dk>`
060978	**Jesper Carlson**	`<carlson@diku.dk>`

3.1.2 Types of interests . . . 5
3.1.3 Power . . . 5
3.1.4 Routing . . . 5
3.2 Design . . . 6
3.2.1 Routing . . . 6
3.2.2 Messages . . . 7
3.2.3 Publish/subscribe . . . 8
3.2.4 Other considerations . . . 8
3.3 Implementation . . . 9
3.3.1 Cache . . . 9
3.3.2 ConstantSensor . . . 10
3.3.3 Sensor . . . 10
3.3.4 NodeConfig . . . 10
3.3.5 NodeList . . . 10
3.3.6 Timestamp . . . 11
3.3.7 DirectD . . . 11

4 Part 3: Testing framework **15**
4.1 Test-driven development . . . 15
4.2 Automated tests and the DIKU Testbed . . . 15
4.3 Testing TinyOS and nesC code . . . 15
4.4 Logging and assertions . . . 16
4.5 The test framework . . . 18
4.6 Test results . . . 19
4.6.1 The test suite . . . 19
4.6.2 Logging . . . 19
4.6.3 SPI . . . 20
4.6.4 Publish/subscribe . . . 21

5 Conclusion **22**

A Test applications **24**
A.1 TestLog . . . 24
A.2 TestRadio . . . 26
A.3 TestDirectD . . . 27
A.4 TestSuite . . . 29

called chip enable or slave select, is used for addressing which device to communicate with by setting it high when transmitting.

There exist two SPI interfaces for TinyOS: ByteSPI and FastSPI. The first interface is intended for slow SPI transmissions and thus uses non-blocking requests via split-phase operations to improve concurrency. The FastSPI interface is intended for SPI devices running with a fast transfer rate and where the implied "blocking" (or waiting for the hardware) is an acceptable trade off for increased speed by avoiding an extra function call. Additionally, the FastSPI interface reduces complexity in the state machine and is more straightforward to use since it closer resembles the transfer mode of SPI.

The registers involved with the setup and use of the SPI hardware is described in Table 1.

Registers in use	
SPIC1	Primary control register, consists of the `SPIE`, `SPE`, `SPTIE`, `MSTR`, `CPOL`, `CPHA`, `SSOE` and `LSBFE` bits
SPIC2	Secondary control register, consists of the `MODFEN`, `SPC0`, `BIDIROE` and `SPISWAI` bits
SPIBR	Baud rate control register, contains 3 `SPPR` and 3 `SPR` bits: the baud rate divisors
SPIS	Status register, contains the `SPTEF`, `SPRF` and `MODF` bits
SPID	Data register, *writes* write to the transmit data buffer and *reads* read from the receive data buffer
PTED	Port E data
PTEDD	Port E data direction, sets whether port E pins are output (1) or input (0)
PTEPE	Port E pull-up enable, decides for input whether pull-up is enabled (1) or disabled (0)

Table 1: **SPI registers.** Description of the SPI registers.

2.1 Initialization

Both driver initialization and shutdown is handled via the `StdControl` interface provided by the SPI driver. The interface requires the driver to implement an init and start command. Since there is no static state to preserve across resets and no sub-components that need to be initialized there is no need for an `StdControl.init`. The main driver initialization can therefore be deferred to the start command. Unfortunately, nesC does not ensure the order in which the `StdControl.start` functions are called. This can cause problems when "starting" the radio with SPI not yet configured. To solve this, we have decided to configure the SPI hardware in a separate function called `initSPI` and call it from both `StdControl.start` and `StdControl.init`.

The purpose of the initialization is to enable the SPI hardware in a configuration that matches the way it will be used. The code to initialize the SPI to match the nodes in the testbed and program is presented below.

```
37        void initSPI()
          {
                  SPIC1 = 0x50;   /* Enable SPI master, set timing edge. */
                  SPIC2 = 0x00;   /* Set to the defaults. */
                  SPIBR = 0x00;   /* Set to highest baudrate. */
42        }
```

In the following we will describe the various register and bit settings. The flags and settings presented below can be found described in [5] in Cchapters 12.3.1 and 12.4.1 unless otherwise specified and a description of the flags can be found in Table 2. The table shows name, value, and a small description.

To enable SPI we need to set `SPE=1` and we set `MSTR=1` since we want the MCU to be master. Furthermore, we need to set `CHPA=0` and leave `CPOL` at default to ensure the timing of the MCU and the radio is synchronized.

SPPR	0	Baud rate prescaler divisor
SPR	0	Baud rate divisor

Bits changed.		
Bit	**value**	**Description**
SPE	1	Enable SPI; makes the MCU allocate the four port E pins
MSTR	1	Sets this device to be a master (MCU)
CPHA	0	First edge on SPSCK occurs at the middle of the first cycle of an 8-cycle transfer

Table 2: **SPI flags** Description of the used values for the SPI flags.

This way, the first edge of the SPSCK occurs at the middle of the first cycle of an 8-clock data transfer. Leaving SPIE = 0 and SPTIE=0 disables interrupts and lets us handle the SPI buffers by polling. Setting LSBFE=0 make SPI data transfers start with most significant bit first. All of the above settings can be achieved by setting the SPIC1 register to $0x50$.

The SPIC2 register can be left as default, see [5] Chapter 12.4.2 for more information. This means that SPI uses separate pins for input and output, due to SPC0=0, which again leaves BIDIROE=0 to be ignored. Since we have disabled interrupts and since we are only communicating with one peripheral, we can leave the mode fault function disabled, i.e. MODFEN=0, which leaves SPIC1's SSOE=0 without influence. Together with MSTR=1, MODFEN=0 ensures that SPI disregards the $\overline{SS}$ pin. Finally, leaving SPIC2 as default sets SPIWAI=0 which leaves the SPI clock operating when the MCU is in wait mode. This seems reasonable since we are not using the MCU power states, however this would be a possible optimization for improving energy consumption.

Since we need the radio to function at full capacity we can leave the SPIBR register as default forcing the highest possible baud rate.

Enabling the SPI will ignore the setting of the PTEDD register for the pins we are going to use, see[5] page 90. Furthermore, it defines what the PTED register is used for, making PTE2=$\overline{SS}$, PTE3=MISO, PTE4=MOSI and PTE5=SPSCK. Since no internal pull-up device for the pins is needed, we can leave PTEPE as default, see [5] page 91.

2.2 Shutdown

The StdControl interface also requires a stop command to be implemented that allows for shutting down access to the device. StdControl.stop can be seen below.

```
        command result_t StdControl.stop()
        {
62              SPIC1 = 0x00;
                return SUCCESS;
        }
```

The stop command simply disables the SPI device by setting the SPIC1 register to zero whereby specifically the SPE flag will be cleared.

```
                /* Write data to the transmit buffer. */
                SPID = data;
85
                /* Wait for the receive buffer flag to clear. */
                while (!SPIS_SPRF)
                        ;

90              /* Clear the status register */
                temp_value = SPIS;

                /* Read the data from the radio and return it. */
                return SPID;
95      }
```

The `txByte` command will wait for a buffer using a busy loop checking the transmit buffer empty flag `SPTEF` in the `SPIS` register. When the flag has been cleared it will write the data into the transmit buffer via the `SPID` register. Then it will wait for the receive buffer to become full as indicated by the `SPRF` flag. This is done by using a busy loop to check that the the flag is set before continuing. Finally, the `SPIS` status register is cleared and the data from the radio is read from the `SPID` register and returned. By reading the `SPIS` registers while `SPTEF` is set and then reading from the `SPID`, we cleanup the `SPIS` register and a new transmission can begin. Alternatively, the cleanup code could be moved to the start making each transmission clean up before communicating. We have chosen that each transmission cleans up before exiting, because this will enable us to catch bugs caused by concurrent access since the second instance entering `txByte` will be caught in the `SPTEF` loop forever.

An alternative to simply doing a while loop would be to measure how long the SPI transmission takes. We could then insert an amount of `NOP` instructions that take the same amount of cycles to complete, eliminating the need for busy waiting. However since we can not accurately measure the time ourselves, we have chosen not to do this.

Depending on whether the driver is going to be used by several components which can access the device concurrently locking may be an issue. This could especially be a problem if several components in the application needs access to the SPI module e.g. if the SPI device was connected to several slave devices. However, since only the radio is wired to the SPI, we assume that only one component will access the SPI device, or that access by multiple components is handled at a higher level. We are aware of the fact that the radio has two components that access SPI, but we assume that the radio handles any concurrency issues internally. Additionally, introducing locking in the FastSPI driver would compromise the emphasis on fast transfers.

topology. This makes matching sources and sinks with the same interest a non-trivial problem. In addition, a topology change might require a re-matching to be done. It is of course possible to flood the network with every message from sources and sinks, but this is very expensive and best avoided as much as possible.

Therefore we need to design a system that can handle matching of interests based on limited local information.

3.1.2 Types of interests

There are several ways to define the interests nodes subscribe to or publish. Sensor data can be generated by the different kinds of sensors with which the network nodes are equipped. It can also be sampled in different intervals and for different lengths of time. If the sensor nodes can determine their own position, it can include geographical information, e.g. subscribing to sensor information from nodes located in a specific area.

Another option is allowing sinks to subscribe to more abstract data events and not simply specify data readings, e.g. "send me a message when the pig oinks" instead of "send me a message with the microphone data every second". Higher level descriptions allow forwarding nodes more options in dealing with the message, such as data aggregation and in-network processing. E.g. averaging data from multiple sources and combining them into one message before it reaches the sink, instead of averaging all the received messages at the sink. This reduces the amount of communication needed.

Therefore we need a robust way of describing the different kinds of interests that can be published/subscribed.

3.1.3 Power

Sensor networks have the unique problem of power supply. Nodes deployed in the field often have no access to external power, and must rely on batteries that are difficult or impossible to replace when they run low. Therefore a low power consumption is essential for ensuring the longevity of the network. Part of this lies in hardware design and overall node operations like doing duty cycling[7], where a node is asleep in low-power mode most of the time, only waking up into high-power mode when needing to communicate or use the sensor. However it has to be considered in application and algorithm design as well. For example, a node that is the theoretical best pick for shortest path routing, might also be very low on power. It is therefore better to avoid routing through it, thus making it able to remain active for as long as possible.

Therefore we need to consider energy-awareness in our design.

3.1.4 Routing

As described earlier, the sensor network has no overall routing information available. But we still need to be able to send messages between nodes that cannot communicate directly. Duty cycling on the nodes (where they have their radios turned off most of the time to conserve power) may impose additional restrictions on the routing, due to nodes not having their radios turned on all the time. With the network not being static, it is also necessary to discover when new nodes have been added, or known nodes disappear.

Therefore we need a routing system that works with limited knowledge and is topology indifferent.

bouncing back and forth, nodes ignore subsequent identical messages. An identical message is one that has already been received, which then due to the delay in travelling a longer path, is received again from another node. The same message received from the same node is not considered identical, as this may simply be two data readings that are the same.

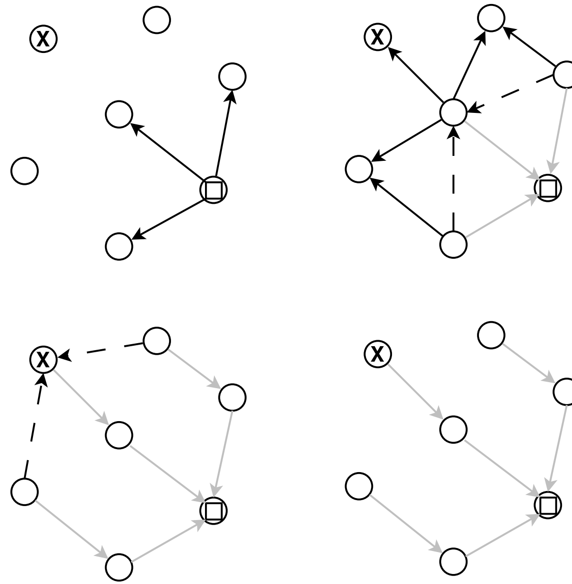

Figure 1: Initial subscription

In Figure 1, the sink marked with a box subscribes to data published by the source marked with an X. Dashed arrows are duplicate messages that are ignored. Grey arrows are gradients, cached interests that will result in data being sent this way. The extra gradients will never have data sent along them, and so will have no effect unless the topology changes.

Note how the gradients in Figure 2 from the two nodes on the top right will not be used in the current situation. But they exist in the eventuality that the network topology changes and they might be needed, instead of having to start from scratch with new gradients.

Once the interest reaches a node that is a source for this kind of data, it starts taking sensor measurements and sending data back. Data is sent to the first node the interest was heard from, which should be the one closest to the sink due to transmission times. This is illustrated in figure 3. When a node receives data, it sends it onwards to the first node from which it heard an interest for that kind of data. Consequently, the shortest path between source and sink will be picked.

If a node that is on the gradient path between a source and sink disappears, it will no longer be possible to route messages this way. To detect that this has happened, a node that sends a message to another node (but not broadcasting), also starts listening if that node forwards the message. If it does not do so within a reasonable airframe, it has probably disappeared, and we need to find a new path. This is then done by broadcasting, so that another node with gradients will hopefully be found nearby. Otherwise an entirely new path will be created. Unfortunatey this was one of the enhancements we did not have time to implement and test.

Figure 2: Here a set of gradients already exists, so the new sink uses those as well.

This system of diffusing interests and data makes it possible for sources and sinks to communicate effectively, even when the overall network topology is unknown.

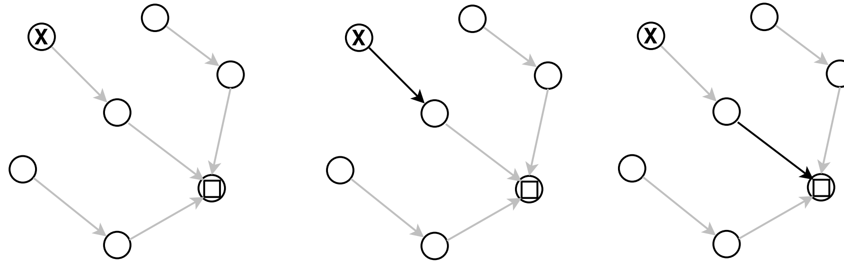

Figure 3: The source sends data back to the sink.

3.2.2 Messages

Our application will need a common message format for communication between nodes. We have chosen to use the existing `TOS_Msg` structure and add the extra fields we need to the data section. This lets us use the existing `TOS_Msg`-based interfaces, without the possible errors of trying to cast our own format into them.

Message structure:

<table>
<tr><td rowspan="2">Address</td><td rowspan="2">Type</td><td rowspan="2">Group</td><td rowspan="2">Length</td><td colspan="5">Data:</td><td rowspan="2">CRC</td></tr>
<tr><td>Sender</td><td>Hops</td><td>Orig sender</td><td>Descriptor</td><td>Sensor data</td></tr>
</table>

Address 16-bit
: The node address of the destination, or a special value for broadcasting. There will often be several nodes in communication range, but some messages are only relevant for one particular node. This allows the other nodes to ignore messages not relevant for them.

Type 8-bit
: The type of the message, this can be either data or interest. Interest messages originate from sinks that want to subscribe to data, while data messages originate from sources sending data.

Group 8-bit
: Magic number to distinguish our nodes from those used by other students. The DIKU Testbed allows several project groups to work with the nodes at the same time. A unique number that identifies our messages will allow us to ignore messages generated by other applications, which might otherwise confuse ours. We picked the number 0x34.

Length 8-bit
: How many bytes of the data block are being used, this will depend on the size of the sensor data.

Data descriptor 8-bit
: The data descriptor this message is an interest or data for.

Sensor data Variable
: Either empty or the sensor measurement, depending on message type. The size of this field is determined by the type of sensor measurement it is from. The length can be calculated from the Length field above, taking into account that the previous fields are 6 bytes long..

CRC 16-bit
: Checksum to detect corrupt packets. We do not use this in our application.

This format covers all the different kinds of messages our nodes will need to exchange.

3.2.3 Publish/subscribe

The data that nodes can subscribe and publish can be specified in many ways. There are constraints on the sensor node itself, such as geographical position, remaining power, or sensor type (temperature, light, etc.). There is also the frequency and number of measurements. The sensor node specification can either be dynamic in almost a kind of query specification where nodes need to be able to determine which of the criteria they fulfil. It can also be determined beforehand with sources programmed to respond to a series of data descriptors.

For our application we settled on a simple approach of specifying a list of data descriptors beforehand. The nodes can then decode these to their matching sensor, sample and period values.

We have decided to keep the publish/subscribe module interface provided with the assignment. However, due to the subscription-driven way our directed diffusion design works, we have not implemented it fully. The explicit publishing and unsubscription command are not needed, when we automatically publish information that a node receives an interest for, and have these subscriptions timeout.

3.2.4 Other considerations

Our system is a little different from the directed diffusion described in [7] and [1]. We use previous data and interest messages as implied gradients and only use these two kinds of messages. This approach is simpler and thus easier to implement at first and then iteratively refine. However, we still believe that it fulfills the goal of directed diffusion with being data-centered and not needing any global routing information.

In this design the transfer of data is entirely controlled by the sinks pulling in data by sending out interests. Another possibility would be to allow sources to actively publish data, even without having heard any interests for it. This changes the focus of the system from retrieving data to instead disseminating or distributing data that might be used where it is overheard. Another way of thinking about this, is whether the system is push or pull centered. We feel that the first approach is a better fit for the kind of system we are designing.

There are different approaches to how subscriptions stop. One is an explicit unsubscription message which will tear down the gradients pointing towards that sink, and eventually make the source stop sending more data. We have chosen instead to have all subscriptions contain an implicit timeout encoded in the data descriptor as the number of data samples desired. If the sink wants more data, it can send a new interest. This both mimics a kind of reinforcement and solves the problem of a source continuously sending data with no idea whether the sink that wanted it is still alive.

Energy awareness in our application is another issue to consider. Our nodes should take into account energy levels as part of the decisions they take. The easiest approach is for nodes low on energy to ignore messages they ought to have forwarded. This effectively makes them drop out of the network most of the time, which the

3.3 Implementation

To make publish/subscribe work in a sensor network, we have implemented a directed diffusion system. This system is assembled from several TinyOS modules. It would have been possible to implement the application as one large module, which would have saved time and reduced complexity in creating interfaces between the modules. However the downside of it being much less modular and much harder to test makes the first approach better. The modules we have created will be described below.

3.3.1 Cache

We need to be able to store messages received from other nodes. To do this we created a `CacheM` module. It stores received messages, enabling us to detect duplicates and check for previous messages. The interface allows for the storage of messages, for checking if a given message already exists in the cache, and for retrieving the gradients these messages represent.

The cache itself consists of an array of messages, with the number of times they have been stored attached. This can obviously be implemented more efficiently with e.g. a tree structure, but this is good enough for our purpose. The modular structure of the program makes a later change easy to implement.

If a program attempts to store too many messages in the cache, previous messages are overwritten, starting with those that have been received the fewest amount of times. The storage of messages can be seen in the following code snippet.

```
command result_t Cache.storeMsg(TOS_MsgPtr msg)
{
        cacheEntry *currEntry;

83      // the position in the cache to be overwritten
        uint8_t overwritePos = 0;
        uint8_t overwriteStores = (uint8_t) -1;

        uint8_t i;
88
        // to make sure we don't overwrite the same position every
            time,
        // rotate the starting element
        rotate = (rotate + 1) % CACHE_SIZE;

93      for (i = 0; i < CACHE_SIZE; i++)
        {
                currEntry = &cache[(i+rotate) % CACHE_SIZE];

                // is the message we want to store in the cache
                    already?
98              if (cache[i].stores > 0 &&
                    msgEqual(&currEntry->msg, msg))
                {
                        currEntry->stores++;
```

```
                overwriteStores = currEntry->stores;
            }
        }

        memcpy(&cache[overwritePos].msg, msg, sizeof(TOS_Msg));
        cache[overwritePos].stores = 1;

        return SUCCESS;
    }
```

3.3.2 ConstantSensor

The motes on the DIKU Testbed have not yet been equipped with sensors. But we need sensor data to publish and subscribe to. To provide this we have created a dummy sensor module called `ConstantSensorM`, that returns a constant value instead of reading from a real sensor attached to the node. It implements the `ADC` interface for requesting and receiving data like a real sensor would.

```
    task void returnData()
    {
        // Return a constant value for every sensor measurement
        signal ADC.dataReady(1);
    }
```

3.3.3 Sensor

In our application, nodes will very often want to subscribe to periodic sensor readings. To make this easier to work with, we have created a sensor module called `SensorM` that encapsulates the timer needed to manage this. The interface allows for the request of a certain data descriptor. When returning data, it is delivered in a format compatible with the `PublishSubscribe` interface. This makes the code in the main module clearer and more concise.

3.3.4 NodeConfig

Our nodes need to know which kinds of sensors they are equipped with, and which tasks they have to perform. To do this, we have created the `NodeConfigM` module. Its interface allows a node to look up if it is a source or sink for a particular data descriptor. If we were using nodes equipped with real sensors, the modular nature of this would allow it to be replaced with a new version that actually detects if the sensor hardware is present. An additional benefit of this module is the ability to configure which nodes are within communication range. When testing, this can be used to simulate the network topology more accurately, since we do not have access to physically move the motes in the DIKU Testbed around.

3.3.5 NodeList

We need to remember which nodes we have heard from recently, so we know if we can communicate with them. To maintain this list we created the `NodeListM` module. Its interface allows for the addition of a node we have

```
	{
		uint16_t diff;

		// We have wrapping.
		if (a > b) {
			diff = b + 1 + (((uint16_t) -1) - a);
		} else {
			diff = b - a;
		}

		return diff;
	}
```

Here we make sure that comparing timestamps works even if the time counter has wrapped around.

3.3.7 DirectD

To tie all these other modules together and implement our directed diffusion logic, we created the `DirectDM` module. It implements the `PublishSubscribe` interface provided with the assignment. By relying on the previously constructed modules, our design can be coded cleanly and readably. An overview of the state machine used by the module is given in the state graph in Figure 4. The states in the upper left and right corner correspond to the subscribe command and data received event in the `PublishSubscribe` interface. States in the lower half of the graph correspond to the handling of interest and data messages.

The most important part of the module is the following message routing code.

```
	event TOS_MsgPtr Receive.receive(TOS_MsgPtr msg)
	{
		uint8_t dataDescriptor;
		directd_msg_t *directd;

		// Ignore if this message is not from our application
		// group or addressed to this node.
		if (msg->group != DIRECTD_GROUP ||
		    (msg->addr != TOS_LOCAL_ADDRESS &&
		     msg->addr != TOS_BCAST_ADDR))
			return msg;

		directd = (directd_msg_t *) msg->data;

		// To aid in testing, we make it so that some nodes can't
		// see each other, even though they're in communication
		// range
		if (call NodeConfig.isNeighbor(directd->sender) == FALSE) {
			LOG_INFO0("message ignored");
			return msg;
		}

```

```
        // Add the message to the cache
        call Cache.storeMsg(msg);
175
        dataDescriptor = directd->dataDescriptor;

        if (msg->type == DIRECTD_INTEREST) {
                // Are we a source for this interest
180             if (call NodeConfig.isSource(dataDescriptor)) {
                        LOG_INFO1("Starting␣sensor␣for␣%u",
                            dataDescriptor);

                        // Start the sensor with the requested
                        // descriptor
185                     call Sensor.startPeriodic(dataDescriptor);
                } else {
                        LOG_INFO0("forwarding␣interest");
                        // Forward it instead
                        // Broadcast if nowhere specific to send to
190                     memcpy(&forwardMsgBuffer, msg, sizeof(*msg));
                        post forwardMsg();
                }

        } else if (msg->type == DIRECTD_DATA) {
195             // Are we a sink for this data type
                if (call NodeConfig.isSink(dataDescriptor)) {
                        signal PS.dataReceived(directd->data,
                                               msg->length -
                                                   DIRECTD_HEAD_SIZE,
                                               dataDescriptor);
200             } else {
                        LOG_INFO0("forwarding␣data");
                        // Forward it instead
                        // Do nothing if nowhere specific to send to
                        memcpy(&forwardMsgBuffer, msg, sizeof(*msg));
205                     post forwardMsg();
                }

        } else {
                LOG_ERROR0("Unknown␣DirectD␣message␣type");
210     }

        return msg;
    }
```

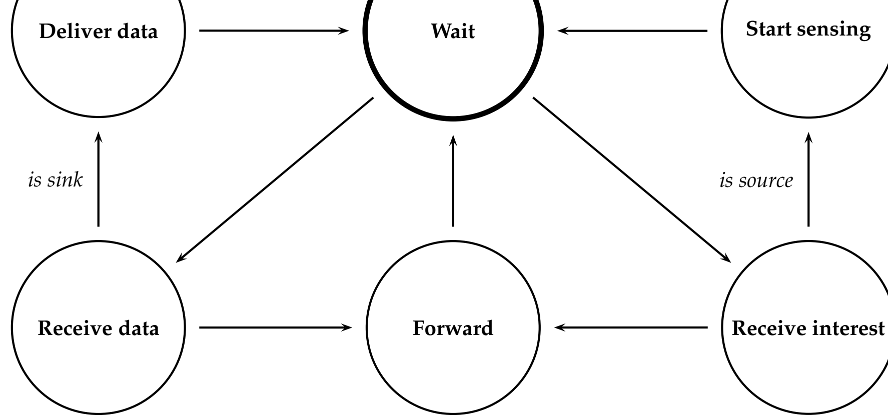

Figure 4: **The DirectDM state graph.** The main state is indicated by a bold circle. Dotted arrows shows related state transitions, e.g. sending an interest will result in data delivery.

To simplify the use of our directed diffusion implementation in applications we have created the `DirectDC` configuration component. Its component graph is depicted in Figure 5. `DirectDC` wires our modules together, as well as components for the radio. This means that our application is intended to run as the sole program on a mote, not as a service for another program. An impression of how an application might use the component can be seen in Figure 9 on page 27.

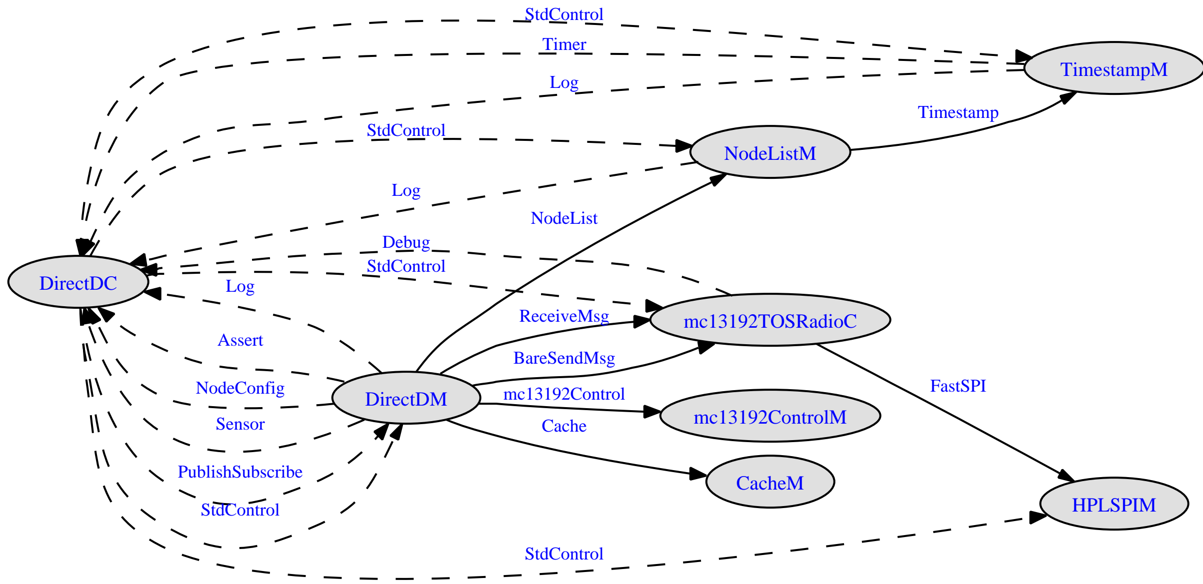

Figure 5: The DirectDC component graph.

just added didn't break any existing functionality. You quickly arrive at the conclusion that it is hard to ensure that any kind of testing will rigorously test the complete system. Testing ends up becoming an overwhelming task at best and pointless at worst.

The solution is to use test-driven development and develop the test framework in parallel with the code[6]. By gradually creating and expanding a test suite the cost of testing is amortized over the entire development process. More important, most of the testing is done at a stage where bug fixing is easy and cheap. Additionally, having a test suite makes the code more *agile*; allowing regular refactoring and even big changes late in the development process because the test suite will help to quickly identify issues[3]. This is crucial for embedded systems since they are often long-lived and may need major upgrades, e.g. if a chip or other hardware component has gone out of production. Finally, because regressions are discovered much faster, testing gives confidence to developers, which is invaluable especially late in the development process when things get hectic.

While the many advantages of test-driven development are hard to deny, there are different considerations to make. While testing embedded system resembles normal testing practices, issues important in the domain of embedded systems affects testing. The most important of these issues are limited resources and timing constraints[4]. Simulating the intended workload can be very hard and it may therefore be necessary to accept that no test framework can fully test certain parts of the code. Another important consideration regarding the test framework is to make clear what approach to testing integrates best with the development environment, which is the topic of the following section.

4.2 Automated tests and the DIKU Testbed

The main goal of the test framework is to make it trivial to test key components. By making it easy to test, code regressions can be discovered almost immediately, which will make it easier to pin-point the change responsible for introducing the regression. A big part of making testing easy is to automate it. This is the approach used in the TinyOS regression testing framework[2], where a series of scripts running on a PC will program two motes with various applications and capture the messages they send and verify them.

Unfortunately, the Re-Mote client for the DIKU Testbed does not support automated access to the motes, neither to program them nor to grab output from the UART. When possible, it is therefore better to batch many tests by running them using one big test suite application to maximize coverage. Because all compilation has to go through the compile host which has proven to add considerable time, batching tests using this approach can also help to decrease the time of the code-compile-test cycle.

Another restriction regarding the use of a remote testbed lies in having no physical access to the motes. It limits the choices available for debugging and testing to those who can be managed via the console. This can especially be a problem for driver development, where access to the low-level details, such as provided by a debugger or oscilloscope, can be essential. Having to use debug printing in order to trace the program flow can be very time consuming and in some cases impossible due to the limited baud rate of the console UART

4.3 Testing TinyOS and nesC code

An important part of testing is to test as small a part of the system as possible at a time in order to have more control of what is being tested[4]. For example, the core components of the system should be tested before testing those that depend upon them, so that regressions are detected at the right level.

The programming model of TinyOS and nesC encourages heavy use of components to modularize the code. This eases testing, since the code to some extent is already separated into small testable parts. However, for complex components it may be necessary to decrease the test granularity and only test *one interface* at a time

Table 3: **Printf-identifiers supported by the log interface**.

rather than *one component*. Dividing up tests based on interfaces makes it possible to reuse the test suite for testing different components providing the same interface and is preferable.

For some components it may be necessary to use stubs to further isolate them and have control over the test environment[4]. This can be useful especially for complex component graphs, since it allows some of the testing to be performed on a PC using a simulator, such as TOSSIM. The nesC language facilitates the use of stubs very well by making it easy to wire in the stub component in place of the original one. One problem may arise when the design principal of *cross component optimization* is in use, since it may be harder to isolate a specific component. Because we have tried to keep each component simple, we have chosen not to use stubs but instead test whole components using a bottom-up approach.

Testing timing constraints is as important as testing functional behavior for an embedded system[4]. This is very hard given the available development environment since it can require that a certain amount of accuracy can be ensured. However, for some components it is is good enough to be able to perform longer running tests, e.g. to test timeouts. The test framework as presented above focuses on testing components separately in a sequential manner in order to eliminate problems. It will therefore not be able to detect defects such as timing errors that happen when all components are running and in use concurrently

Before describing the test suite and results we will first look at the developed logging and assertion interface.

4.4 Logging and assertions

The assignment framework contains a simple interface for printing debug strings. The `Debug` interface provided by the `ConsoleDebugM` component together with the `DBG_STR` set of macros allows strings and integers to be printed to the UART. Compared to printf it is not very flexible and mostly suited for light-weight low-level debug printing. To have a more high-level interface available, a log component has been developed.

The primary interface provided by the log component is the `Log` interface, which gives easy access to printing to the console using printf-like format strings. The format identifiers supported by the logging interface is a trimmed down version of those supported by printf. They are listed in Table 3. All calls to the log module are intended to be made using a set of macros. This slightly violates the TinyOS design by introducing global identifiers, however it is also used by the `dbg` functionality in the TinyOS 1.x core, so it is deemed acceptable. The main reason for using macros is to make it easy to disable all logging at compile-time without changing the code. The following code gives an example how logging is used in the `TestM` component:

```
92        event void TestUnit.done[uint8_t test]()
          {
                  group_index++;
                  if (group_failed) {
                          LOG_INFO2("=> failed %u out of %u test(s)\n",
97                                group_failed, test_number);
                  } else {
                          LOG_INFO1("=> passed all %u test(s)\n", test_number);
                  }
                  post runTests();
102       }
```

all platforms support variadic functions or macros, including the platform used in the DIKU Testbed. The main problem is therefore to find a more portable way to handle a variable number of arguments. By making the caller denote the number of arguments at the call site, logging is made a little less flexible compared to printf. However, it has the advantage of making it possible to check during formatting that the number of passed arguments corresponds to the number of format identifiers.

Since there is no `va_list` type available, arguments are passed to the log component via an array. All argument are first casted to an `uint32_t` value and then casted back to their original type based on the format identifiers. Using a 32-bit value ensures that most types and values available on the platform can be handled correctly. To give an idea of the overhead involved with the logging interface, the following code snippet shows how arguments are packed into an array before being handed over to the `Log.put()` command:

```
#define LOG_PUT3(level, fmt, arg1, arg2, arg3) \
        { \
                uint32_t argv[3]; \
                argv[0] = (uint32_t) (arg1); \
                argv[1] = (uint32_t) (arg2); \
                argv[2] = (uint32_t) (arg3); \
                call Log.put(LOG_LEVEL_##level, fmt, argv, 3); \
        }
```

Using 32-bit types so extensively can be expensive on the 8-bit architecture of the motes in the DIKU Testbed. If the accuracy of supporting 32-bit types is not required, it should be easy to downgrade logging to only handle 16-bit types. However, as already mentioned the logging interface is designed to be high-level.

Besides printing, the `Log` interface also has the notion of log levels for tagging the severity of log messages. There are only 4 levels to keep it simple ranging from debug messages to error messages. The different log levels and their intended usage are summarized in Table 4. The default log level can be changed at run-time which will cause all log messages of less severity to be ignored. Although not something most application will make use of, it can be essential to avoid concurrent access to the logging to clutter important message or to rate limit the log messages, so that warning and error message which are not expected to be very frequent are ensured to be visible by disabling debug and informational messages for a period of time.

Finally, the log component makes it possible to toggle whether each message being logged should be on a separate line or are part of the same message. This is useful when values from an array or other scalar type should be summarized via an iteration. It is used for implementing a simple test progress indicator in the `TestM` component which displays a small animation. Since the indicator is only meant to be visible until the test result has been determined, the cursor should not be advanced to the next line, since a future message is expected to overwrite the progress string:

```
        command void TestControl.tick()
        {
                call Log.setLineBased(FALSE);
                LOG_INFO2("\r[%u] %c", test_number,
                        ticks[test_ticks++ % (sizeof(ticks) - 1)]);
                call Log.setLineBased(TRUE);
                test_ticked = TRUE;
        }
```

In addition to the `Log` interface, the log component also supports a simple `Assert` interface providing a single command which should be called in case of an assertion failure. Assertions are a great way to embed

expression, a message detailing the expression and the position in the code (in the form of the file name and line number) will be printed in case of failure. As for the log macros there is also defined an assert macro, which will call the assertion failure command with the correct arguments:

```
#define assert(expr) \
        if (!(expr)) { call Assert.failed(#expr, __FILE__, __LINE__); }
```

Logging and assertion handling is provided by the LogM module. All the actual formatting is handled by the commands in the `ConsoleOutput` interface provided by `ConsoleC`. For environments where use of the `Log` and `Assert` interfaces is mixed with use of the `Debug` interface a `LogC` component has been created. It provides all the various console output interfaces: `Debug`, `Log`, and `Assert`.

4.5 The test framework

A test framework has been developed based on some of the above consideration. It has been the goal to provide a simple and very flexible framework. The basic idea is to divide tests into a number of groups each focused on testing a small part of the code, such as testing either a specific interface or all interfaces provided by a component.

Each group of tests is expected to reside inside its own test module, which is required to provide the `TestUnit` interface. This interface allows testing to be started via a *run* command and test completion to be reported back to the main component of the test framework via a *done* event. Besides scheduling when test groups are run, the test framework does not dictate how a test module should arrange and run its group of tests. In other words, a test is handed over the control of the system and is free to manage its own level of concurrency.

For managing the individual tests performed by a test module a `TestControl` interface is provided, which has commands for starting and ending tests as well as reporting progress for longer running tests. Below is an example of its usage. It is taken from the `TestConstantSensorM` component of the `TestSuite` application. Note that the `TestControl` interface has been aliased to `Test` to make the code less verbose:

```
        task void testConstantValue()
        {
                call Test.start("Reading␣constant␣value", SUCCESS);
                samplesLeft = call NodeConfig.getSamples(NODECONFIG_A_ID);
44              call Sensor.startPeriodic(NODECONFIG_A_ID);
        }

        task void checkConstantValue()
        {
49              call Test.tick();
                assert(sensorValue == 1);
                samplesLeft--;
                if (samplesLeft == 0) {
                        call Test.done(SUCCESS);
54                      post testDone();
                }
        }

        event result_t Sensor.dataReady(uint8_t data[], uint8_t size, uint8_t
            dataDescriptor)
```

data has the expected value. Finally, it handles the test end condition, where `Test.done` is called with the test result followed by the next test task being posted.

To simplify verification of return values, the test framework also provides the `Assert` interface. This allows tests to avoid specifically tracking error conditions by simply using the `assert()` macro. If an assertion fails, even one in a component indirectly being tested, the test framework will report the test as failed.

The test output is very minimalistic and will for successfully running tests print a minimum of information. This ensures that it is easy to check the error log and locate failures. Each test group, as indicated by a call to the `group` command of the `TestControl` interface, is reported in a separate section. For each test in the group, the test number is reported and whether it failed or succeeded. Any intermediate progress reported will not be visible in the final output.

In the following example, output from testing the `SPI` and `Timestamp` modules is shown. As can be seen an assertion failure will both show the assertion message complete with file and line number as well a causing the current test to fail. Each test group is summarized at the end.

```
Testing TestSPIM
assert(PTED == 0x04) failed at TestSPIM.nc:42
[1] failed: PTE register values
[2] ok
[3] ok
[4] ok
=> failed 1 out of 4 test(s)

Testing TestTimestampM
[1] ok
[2] ok
=> passed all 2 test(s)
```

4.6 Test results

In the following, the results of all tests are summarized. Information about the test applications, such as component graph and test output, can be found in Appendix A.

4.6.1 The test suite

The test framework described above is primarily used by the TestSuite application. The test suite has test modules dedicated to testing all the main components, including the `TestM` module itself. For example, the `CacheM` module is tested by the `TestCacheM` module. The various test components and what they test for are summarized in Table 5. The component graph of the TestSuite application can be seen in Figure 10 on page 30.

The output of running the TestSuite is available in Appendix A.4 on page 29. As can be seen all tests in the test suite are passed successfully. We conclude that that modules covered by the test suite works correctly.

4.6.2 Logging

The TestLog application has been used for testing logging. Like all the tests apart from the TestSuite application, it does not use the test framework presented above, since it is not possible to easily test text outputted to the

IsSink	Is the node correctly reported as sink
GetPeriod	Are the sampling periods associated with data descriptors correctly reported
GetSamples	Is the sample count associated with data descriptors correctly reported
TestNodeListM	
Add	Check if an added node exists even if others are added afterwards
Timeout	Test that nodes are not reported as existing after a timeout
Full	Can the node add more than the maximum number of nodes to the list
TestSPIM	
SPIC1	Test that the `SPIC1` register is setup as intended
SPIC2	Test that the `SPIC2` register is setup as intended
SPIBR	Test that the `SPIBR` register is setup as intended
TestTestM	
Success	Is success reported if a test was successful
Failure	Is failure reported if a test failed
Assert	Does a failed assert mark a test as failed
AssertFailure	Is failure reported if there is a failed assert and a failure report
AssertMacroTrue	Does a the assert macro trigger a failure if the expression passes true
AssertMacroFalse	Is failed reported if an expression passes false
TestTimestampM	
Incremental	Test that periodical requests for a timestamp give increasing values
TimeDiff	Test various differences between timestamps as reported by `Timestamp.diff`

Table 5: **Test suite.** Description of the different components in the TestSuite and their individual tests.

UART. Instead, it requires visual verification by the tester. The component graph of the TestLog application can be seen in Figure 7 on page 24.

TestLog is divided into different test groups each devoted to one part, such as testing a specific printf identifiers. Each line denoted by `=>` is a separate test. For most of the tests, the expected output is printed in the start of the line, followed by a `=`, followed by the actually formatted output. However, some of the tests, such as printing of the NUL char this is not the case. The output of the TestLog is available in Appendix A.1 on page 24 and is as expected.

4.6.3 SPI

Two applications have been used for testing the SPI driver. First, the `TestSPIM` module of the `TestSuite` application tests that the 3 SPI control registers has the expected values. As can be seen, the `TestSPIM` section of the `TestSuite` output available in Appendix A.4 on page 29 shows that the three SPI register tests all pass. This confirms that the SPI driver correctly sets up the SPI hardware.

As for the correctness of SPI transmission, the `TestRadio` application has been used for testing. The component graph of the `TestRadio` application is presented in Figure 8 on page 26. It is intended to be run on two or more nodes in the testbed. Each node periodically sends a message with a sequence number and small payload and prints information about messages received from neighboring nodes. The output of the `TestRadio` application in Appendix A.2 on page 26 shows a test where node 5 is started first and is running for the entire test,

compile to the architecture used on the DIKU Testbed motes we use a centralized server with the only complier license. This server has been inaccessible for approximately one third of the time we have worked on the project, including most of the last week before the deadline. To further complicate things, the DIKU Testbed itself has had problems. Many of the motes in the testbed have worked erratically or not at all throughout the assignment period. This has left us with as few as four usable motes to be shared among all the students working on the course. Sadly, this has been quite problematic for our testing and iterative development, only leaving time to get a basic version of our directed diffusion system running, and not allowing it to be tested as thoroughly as we wanted.

To test the routing of interests and data, we set up several test cases. First, a very simple situation with only two nodes, a source and a sink. The sink periodically sends out interests, and displays any data received back. The source responds to any subscriptions by returning data. After verifying that it worked correctly we added code to test multi-hop routing. To test this we made the `TestDirectD` application depicted in Figure 9 on page 27. It assumes a network with three nodes: a source, a sink and an extra node are used. As before the source wants data from the sink, but this time, they are not in communication range of each other. This means that it will have to be routed via the third node. To test this we had to force the sink and source to discard messages received directly from the other due to the fact that all the nodes are located within communication range.

The output from the `TestDirectD` application in appendix A.3 on page 27 shows three nodes, node 4 acting as sink, node 5 as source and node 7 as the node that forwards between node 4 and 5. Node 7, node 5 and then node 4 was started making sure that the source was up and running before the sink subscribed to its data. The output shows that the sink subscribes to the source and the source receives this subscription through the forwarding node. It is worth mentioning that "message ignored" is logged each time the source or the sink receives a message directly from each other. After successfull subscription the source sends sensor data through the forwarding node to the sink. Somehow the last sensor reading is not received by the sink even though the forwarding node claims it has forwarded it. We have not been able to test where the error is due to time constraints. After subscription timeout the sink subscribes again and the output shows a similar result. However, even with this problem we think it is reasonable to conclude that our publish/subscribe system works correctly.

To help with testing we developed a more advanced logging framework that the existing one. This has enabled us to document how we tested the two other parts of the assignment. It has made us confident that the various parts of our implementation works correctly, and shows the importance of unit testing.

Focusing on simplicity, our approach of using small individually testable components to make up a larger whole, has proven a good method for iterative development. In conclusion we feel that despite the difficulties we have encountered, we did end up with a decent finished version of our designs.

Viewed online.

[4] Vincent Encontre.
Testing embedded system: Do you have the guts for it?
Internet: `http://www-128.ibm.com/developerworks/rational/library/459.html`.
Viewed online.

[5] Freescale.
HCS08 microcontroller datasheets.
Freescale, 2004.

[6] James Grenning.
Test driven development.
Internet: `http://www.ganssle.com/tem/tem127.pdf`.
Viewed online.

[7] Feng Zhao and Leonidas Guibas.
Wireless Sensor Networks - An Information Processing Approach.
Morgan Kaufmann, 2004.

Output from the TestLog application.

```
Ã¿esting line based logging
 => multiple strings on one line
Testing log levels
 => debug info warn error = debug info warn error
 => info warn error = info warn error
 => warn error = warn error
 => error = error
Testing LOG_DEBUG
 => a = a
 => ab = ab
 => abc = abc
 => abcd = abcd
Testing LOG_INFO
 => a = a
 => ab = ab
 => abc = abc
 => abcd = abcd
Testing LOG_WARN
 => a = a
 => ab = ab
 => abc = abc
 => abcd = abcd
Testing LOG_ERROR
 => a = a
 => ab = ab
 => abc = abc
 => abcd = abcd
Testing percent escaping
 => %, %, and lots of %, %, and %
Testing %c
 => 'A' == 'A'
 => '\0' == ''
Testing %d
 => 0 == 0
 => -1 == -1
 => 12345 == 12345
Testing %ld
 => 0 == 0
 => -1 == -1
 => 123456 == 123456
Testing %i
 => 25 == 25
Testing %li
 => -123456 == -123456
Testing %u
```

```
Testing %x
 => 0x0010 == 0x0010
 => 0x1024 == 0x1024
Testing %lx
 => 0x00010000 == 0x00010000
 => 0x11223344 == 0x11223344
 => strings == 0x00003497
Testing %p
 => NULL == 0x0000
 => 0x0010 == 0x0010
 => strings == 0x3497
Testing %s
 => (null) = (null)
 => strings = strings
 => static string = static string
 => a b = a b
 => a b c = a b c
 => a b c d = a b c d
Testing %.*s
 => (null) = (null)
 => '' = ''
 => str = str
 => string = strings
Testing complete.
```

Figure 8: The TestRadio component graph

Node 5

```
tarting TestRadio.
Packet #0 sent
Packet #1 sent
Packet #2 sent
Packet #3 sent
Packet #4 sent
Packet #5 sent
Packet #0 from 8 [NEP06_07]
Bad message
Packet #6 sent
Packet #1 from 8 [NEP06_07]
Packet #7 sent
Packet #2 from 8 [NEP06_07]
Packet #8 sent
Packet #3 from 8 [NEP06_07]
Packet #9 sent
Packet #4 from 8 [NEP06_07]
Packet #10 sent
Packet #5 from 8 [NEP06_07]
Packet #1 from 65535 []
Packet #6 from 8 [NEP06_07]
Packet #11 sent
Packet #7 from 8 [NEP06_07]
Packet #12 sent
Packet #8 from 8 [NEP06_07]
Packet #13 sent
Packet #9 from 8 [NEP06_07]
Packet #14 sent
Packet #10 from 8 [NEP06_07]
Packet #15 sent
Packet #11 from 8 [NEP06_07]
Packet #16 sent
Packet #12 from 8 [NEP06_07]
Packet #17 sent
Packet #18 sent
Packet #19 sent
```

Node 8

```
tarting TestRadio.
Packet #5 from 5 [NEP06_07]
Packet #0 sent
Bad message
Bad message
Packet #6 from 5 [NEP06_07]
Packet #1 sent
Packet #7 from 5 [NEP06_07]
Packet #2 sent
Packet #8 from 5 [NEP06_07]
Packet #3 sent
Bad message
Packet #9 from 5 [NEP06_07]
Packet #4 sent
Bad message
Packet #10 from 5 [NEP06_07]
Packet #5 sent
Bad message
Packet #6 sent
Packet #11 from 5 [NEP06_07]
Packet #7 sent
Packet #12 from 5 [NEP06_07]
Packet #8 sent
Packet #13 from 5 [NEP06_07]
Packet #9 sent
Packet #14 from 5 [NEP06_07]
Packet #10 sent
Packet #15 from 5 [NEP06_07]
Packet #11 sent
Packet #16 from 5 [NEP06_07]
Packet #12 sent
Packet #17 from 5 [NEP06_07]
```

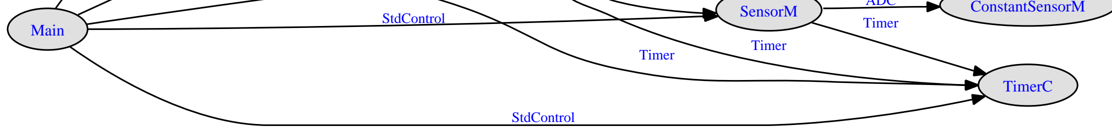

Figure 9: The TestDirectD component graph

```
Subscription 0 returned: 1
message ignored
Subscription 0 returned: 1
message ignored
Subscription 0 returned: 1
message ignored
Subscription 0 returned: 1
message ignored
Subscription 0 returned: 1
message ignored
Subscription 0 returned: 1
message ignored
Subscribing to 0
message ignored
Subscription 0 returned: 1
message ignored
Subscription 0 returned: 1
message ignored
Subscription 0 returned: 1
message ignored
Subscription 0 returned: 1
message ignored
Subscription 0 returned: 1
message ignored
Subscription 0 returned: 1
message ignored
Subscription 0 returned: 1
message ignored
Subscription 0 returned: 1
message ignored
Subscription 0 returned: 1
message ignored
```

```
Sending data
Sending data
Sending data
message ignored
Starting sensor for 0
Sending data
Sending data
Sending data
Sending data
Sending data
Sending data
Sending data
Sending data
Sending data
Sending data
```

```
forwarding data
forwarding data
forwarding interest
forwarding data
forwarding data
forwarding data
forwarding data
forwarding data
forwarding data
forwarding data
forwarding data
forwarding data
forwarding data
```

```
assert(FALSE) failed at TestTestM.nc:116
[7] ok
=> passed all 7 test(s)

Testing TestSPIM
[1] ok
[2] ok
[3] ok
=> passed all 3 test(s)

Testing TestTimestampM
[1] ok
[2] ok
=> passed all 2 test(s)

Testing TestCacheM
[1] ok
[2] ok
[3] ok
[4] ok
[5] ok
=> passed all 5 test(s)

Testing TestNodeListM
[1] ok
[2] ok
NodeList full!
[3] ok
=> passed all 3 test(s)

Testing TestNodeConfigM
[1] ok
[2] ok
[3] ok
[4] ok
=> passed all 4 test(s)

Testing TestConstantSensorM
[1] ok
=> passed all 1 test(s)

Testing complete.
```

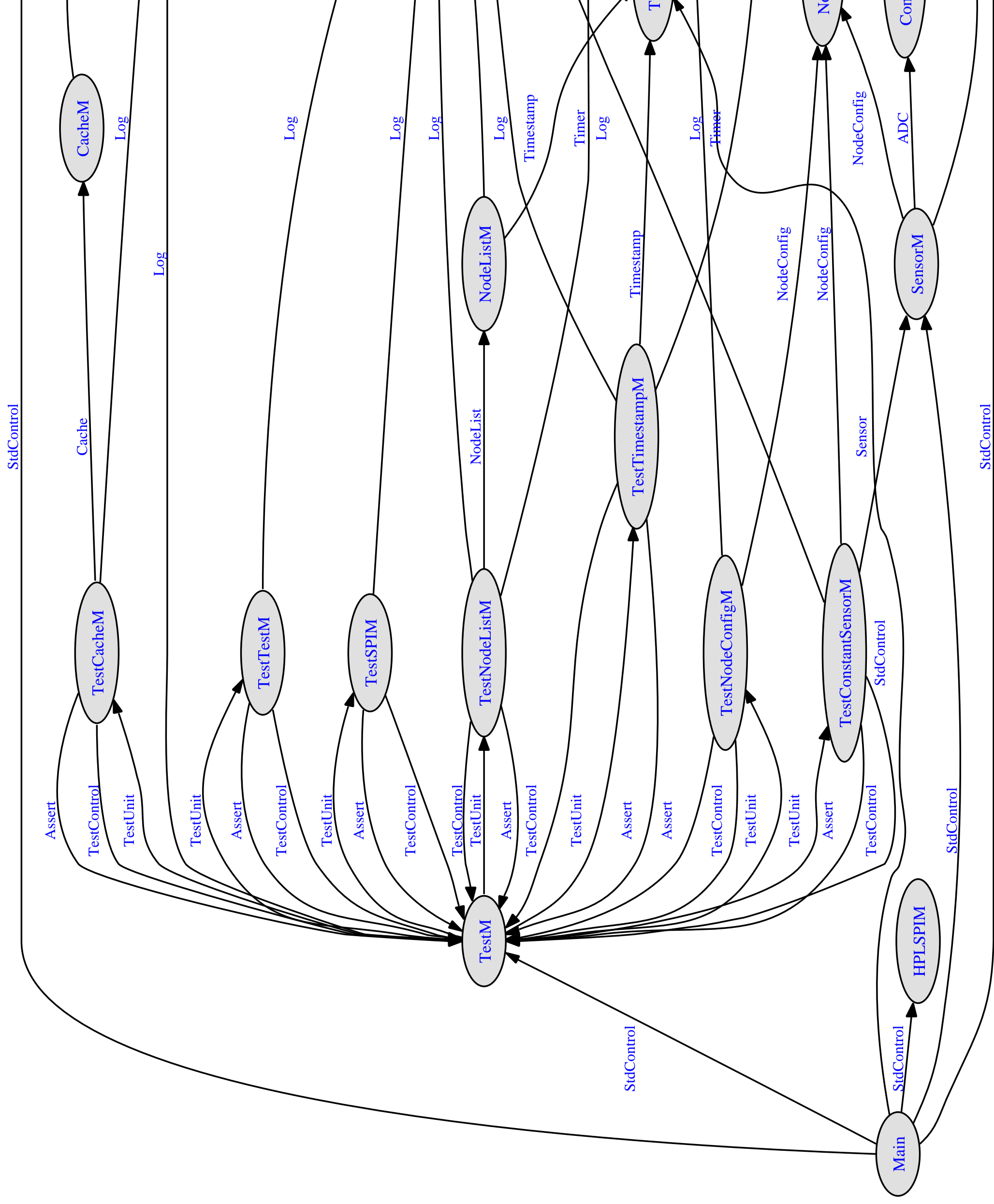

Figure 10: The TestSuite component graph