

## NAME

smacq-modules - pipeline module programming guide

## SYNOPSIS

```
#include <smacq.h>
```

Each module must provide a statically defined structure of type `struct smacq_functions` named `smacq_module_table` such as the following:

```
struct smacq_functions smacq_module_table =
{
    produce: &smacq_module_produce,
    consume: &smacq_module_consume,
    init: &smacq_module_init,
    shutdown: &smacq_module_shutdown,
    algebra: { vector:1, boolean: 1, demux: 1 }
};
```

By convention, the referenced functions also include the name of the module. The API for each function is as follows:

```
static smacq_result smacq_module_init(struct smacq_init * context);
```

```
static smacq_result smacq_module_consume(void * state, const dts_object *, int * outchan);
```

```
static smacq_result smacq_module_produce(void * state, const dts_object **, int * outchan);
```

```
static smacq_result smacq_module_shutdown(void * state);
```

The algebra element is optional and is used only by the dataflow optimizer. The following elements of the algebra structure are as follows: Vector specifies that the module can be used with a single input and a single output, or can be used with a vector of sets of arguments separated by semicolons and a corresponding vector of output channels. Boolean specifies that the module merely filters out some data and can be reordered in the dataflow by an optimizer. Demux specifies that the module demultiplexes output data among multiple output channels. If a demux module fails to set the demux bit, then the optimizer may produce disfunctional output.

```
static struct smacq_options smacq_options[];
```

## DESCRIPTION

SMACQ(1) is an extensible component system for analyzing streams of structured data. This manpage describes the programming API for creating pipeline modules. Type modules are documented separately in `dts-types(3)`.

This document describes the programming interface used by authors of dataflow modules. These modules are dynamically loaded and may be instantiated multiple times. Global and static variables are therefore deprecated for most cases.

Each module must declare a `smacq_module_table` structure of type `struct smacq_functions` referring to static functions described below:

The `init` function is called to initialize a new instantiate of a module. The return value is a module exit

code, with 0 indicating no error. All parameters are passed in a single structure:

```
struct smacq_init {
    int isfirst;
    int islast;
    char ** argv;
    int argc;
    smacq_environment * env;
    void * state;
    smacq_graph * self;
};
```

If any per-instantiation storage is required, it should be allocated in **init** and returned in the state parameter. The **env** element should be saved and passed to any library functions that require an argument of type **smacq\_environment**. Arguments to the module are passed in standard argv, argc form. This structure will be reused after **init** returns, so anything you wish to save out of it must be copied.

The **smacq\_produce()** function is called to output data from the module. It is passed the instantiation's state variable and a pointer in which to store a pointer to a new data object. The return code should be SMACQ\_PASS for success, SMACQ\_END if there is no more data to be produced (ever), or SMACQ\_ERROR in case of error. If there is more data to be produced, the result should be SMACQ\_PASS|SMACQ\_PRODUCE. Produce is called for the first module in a pipeline, as well as right before a shutdown. Other than that, it is only called if the previous call to the module (smacq\_consume() or smacq\_produce()) returned SMACQ\_PRODUCE or SMACQ\_CANPRODUCE. SMACQ\_PRODUCE requires that smacq\_produce() be called before another consume, while SMACQ\_CANPRODUCE may or may not trigger an immediate call to smacq\_produce().

The **smacq\_consume()** function is called when there is new data for a module to process. The return value signals what should be done with the data. The SMACQ\_FREE value says that the data object no longer needed. SMACQ\_PASS specifies that the structure should also be passed to the next module (if there is one) in the pipeline. SMACQ\_ERROR specifies that there was a fatal error consuming the packet. SMACQ\_PRODUCE signals that smacq\_produce() must be called before smacq\_consume() can be called again. SMACQ\_CANPRODUCE says that smacq\_produce() will return data, but does not have to be called for smacq\_consume() (presumably your module is doing some kind of buffering in this case).

Both **smacq\_consume** and **smacq\_produce()** may fill in the outchan parameter if they wish to limit the flow of data to a specific child in the data-flow graph. Otherwise, the default value of -1 will cause data to go to all children.

The **smacq\_shutdown()** function is called when there is no more data to process. It is responsible for freeing and resources used by the module. The return value is an exit code, with 0 indicating no error.

## LIBRARY

See the **dts(3)** documentation for information on using dts\_object objects.

**int smacq\_getoptsbyname(int argc, char \*\* argv, int \* argc\_left, char \*\*\* argv\_left, struct smacq\_options \* options, struct smacq\_optval \* optvals)**

Parse the argv argument vector according to the legal options specified in the *options* array and store the values in the memory locations pointed to by the *optvals* array. Each array is terminated with a structure with a NULL name.

```

struct smacq_options {
    char * name;
    smacq_opt default_value;
    char * description;
    smacq_opt_type type;
    int flags;
};

```

```

struct smacq_optval {
    char * name;
    smacq_opt * location;
};

```

Valid types are: *SMACQ\_OPT\_TYPE\_INT*, *SMACQ\_OPT\_TYPE\_USHORT*, *SMACQ\_OPT\_TYPE\_TIMEVAL*, *SMACQ\_OPT\_TYPE\_UINT32*, *SMACQ\_OPT\_TYPE\_BYTES*, *SMACQ\_OPT\_TYPE\_STRING*, *SMACQ\_OPT\_TYPE\_UBYTE*, *SMACQ\_OPT\_TYPE\_DOUBLE*, *SMACQ\_OPT\_TYPE\_BOOLEAN*

## OUTPUT QUEUES

It is often necessary for modules to queue data objects for output. The following routines enqueue and dequeue objects. The queue is a struct *smacq\_outputq* \* and is initialized to NULL.

```
void smacq_produce_enqueue(struct smacq_outputq ** qp, const dts_object * o, int outchan)
```

```
smacq_result smacq_produce_dequeue(struct smacq_outputq ** qp, const dts_object ** o, int * outchan)
```

```
smacq_result smacq_produce_canproduce(struct smacq_outputq ** qp)
```

## HASH TABLES

It is convenient to use iovec hash tables in DTS modules. See the *bytesthash(3)* manpage for more information.

## THREAD SHIM

The native module API described above is based on event-driven callbacks. However, a module can instead have its own thread and a read/write API from a while loop. (Note that a module thread may be implemented as a non-preemptive co-routine). To use a thread, the module function table should be initialized as follows:

```
struct smacq_functions smacq_module_table = SMACQ_THREADED_MODULE(smacq_module_loop)
```

The *smacq\_module\_loop* function can use the following functions:

```
const dts_object * smacq_read(struct smacq_init * context)
```

Returns a *dts\_object* or NULL if there are no more objects to be read the loop should return.

**void smacq\_write(struct state \* state, dts\_object \* datum, int outchan)**

**void smacq\_decision(struct smacq\_init \* context, const dts\_object \* datum, smacq\_result result)**

**int smacq\_flush(struct smacq\_init \* context)**

Returns 0 normally, or 1 when no more objects can be written and the caller should return.

## **DYNAMIC ARRAYS**

**void darray\_init(struct darray \* darray, int max\_hint)**

Initialize the dynamic array based on the hint specifying the maximum number of elements expected.

**void \* darray\_get(struct darray \* darray, int element)**

Return the specified element of the array.

**void darray\_set(struct darray \* darray, unsigned int element, void \* value)**

Set the specified element of the array to the given value.

**void darray\_free(struct darray \* darray)**

Free all data associated with the array.

## **SEE ALSO**

**smacq(1), dts(3) bytewidth(3)**