endace
accelerated

Embedded Messaging API Programming Guide

EDM04-15

Protection Against Harmful Interference

When present on equipment this manual pertains to, the statement "This device complies with part 15 of the FCC rules" specifies the equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the Federal Communications Commission [FCC] Rules.

These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment.

This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications.

Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at their own expense.

Extra Components and Materials

The product that this manual pertains to may include extra components and materials that are not essential to its basic operation, but are necessary to ensure compliance to the product standards required by the United States Federal Communications Commission, and the European EMC Directive. Modification or removal of these components and/or materials, is liable to cause non compliance to these standards, and in doing so invalidate the user's right to operate this equipment in a Class A industrial environment.

Disclaimer

Whilst every effort has been made to ensure accuracy, neither Endace Technology Limited nor any employee of the company, shall be liable on any ground whatsoever to any party in respect of decisions or actions they may make as a result of using this information.

Endace Technology Limited has taken great effort to verify the accuracy of this manual, but nothing herein should be construed as a warranty and Endace shall not be liable for technical or editorial errors or omissions contained herein.

In accordance with the Endace Technology Limited policy of continuing development, the information contained herein is subject to change without notice.

Published by:

Endace Technology® Ltd
Level 9
85 Alexandra Street
PO Box 19246
Hamilton 3244
New Zealand
Phone: +64 7 839 0540
Fax: +64 7 839 0543
support@endace.com
www.endace.com

International Locations

New Zealand
Endace Technology Limited
Building 7, Lambie Drive
PO Box 76802
Manukau City 2241
New Zealand
Phone: +64 9 262 7260
Fax: +64 9 262 7261

Americas
Endace USA® Ltd
Suite 220
11495 Sunset Hill Road
Reston, Virginia 20190
United States of America
Phone: +1 703 382 0155
Fax: +1 703 382 0155

Europe, Middle East & Africa
Endace Europe® Ltd
Sheraton House
Castle Park
Cambridge CB3 0AX
United Kingdom
Phone: +44 1223 370 176
Fax: +44 1223 370 040

Copyright 2006-2007 Endace Technology Ltd. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the Endace Technology Limited.

Endace, the Endace logo, Endace Accelerated, DAG, NinjaBox and NinjaProbe are trademarks or registered trademarks in New Zealand, or other countries, of Endace Technology Limited. All other product or service names are the property of their respective owners. Product and company names used are for identification purposes only and such use does not imply any agreement between Endace and any named company, or any sponsorship or endorsement by any named company.

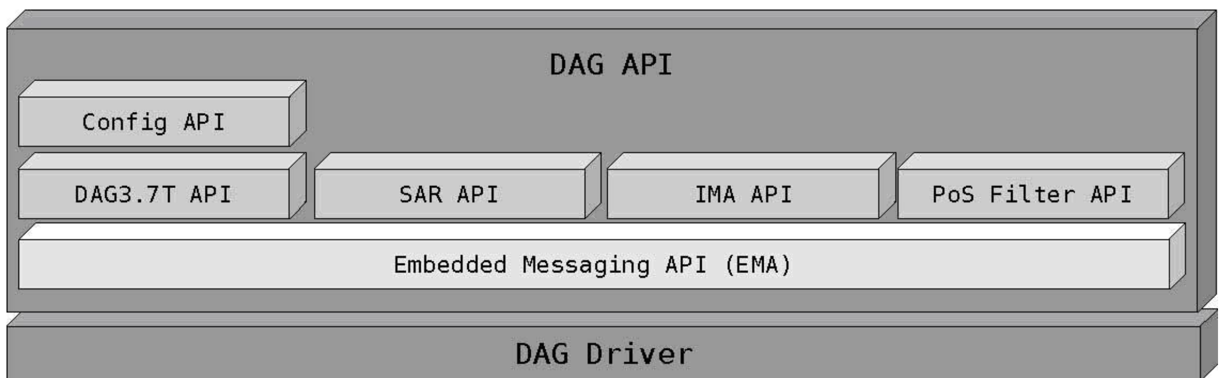
Use of the Endace products described in this document is subject to the Endace Terms of Trade and the Endace End User License Agreement (EULA).

Contents

Introduction 1
Overview of the Embedded Messaging API (EMA) ... 1
Embedded Processor Reset ... 1
Naming Convention ... 2
Usage Example ... 2
Function Definitions 3
dagema_le16toh Function ... 3
dagema_le32toh Function ... 3
dagema_htole16 Function ... 3
dagema_htole32 Function ... 4
dagema_get_last_error Function ... 4
dagema_open_conn Function ... 5
dagema_close_conn Function ... 6
dagema_reset_processor Function ... 7
dagema_reset_processor_with_cb Function ... 8
dagema_send_msg Function ... 9
dagema_recv_msg Function ... 10
dagema_recv_msg_timeout Function ... 11
dagema_set_msg_handler Function ... 12
Version History 13

Overview of the Embedded Messaging API (EMA)

The EMA is a low level library that provides a standard interface to DAG cards that have embedded network processors (currently this includes the DAG 3.7T and DAG 7.1S). The EMA sits below the various API's used to interface to specialist functionality provided by the software running on the embedded network processors.



Layout diagram of Endace APIs

Some of the above libraries (SAR API, IMA API, etc) that depend on the EMA library also depend on an open EMA connection to work correctly, refer to the documentation for the specific library in use to determine requirements.

The EMA provides an interface analogous to that of a serial port; to send a message you must first open a connection, only one process can have an open connection to a DAG card at a time and once finished sending/receiving messages the connection should be closed.

In its simplest form there are four functions that are required to send and receive messages to the embedded processor

- `dagema_open_conn`: Opens a connection to a particular DAG card, this must succeed in order to send or receive messages.
- `dagema_close_conn`: Closes an open connection, all connections should be closed prior to terminating the application.
- `dagema_send_msg`: Sends a message to the embedded network processor.
- `dagema_recv_msg` & `dagema_recv_msg_timeout`: Waits for a message to be received from the embedded network processor and then copies the received message into a user supplied buffer. The `dagema_recv_msg` function blocks indefinitely until a message is received, `dagema_recv_msg_timeout` blocks until either a message is received or the supplied timeout value has elapsed.

Embedded Processor Reset

Two functions are provided to reset the embedded network processors, `dagema_reset_processor` and `dagema_reset_processor_with_cb`, each perform the same function, but the later expects a pointer to a callback function that will be called with status information during the processor reset process. Typically a reset should be performed when an application first starts, prior to attempting to open a connection.

Naming Convention

All functions in the API are prefixed with `dagema_`. The arguments for the functions are fixed to the C99 standards (uint8_t, uint16_t, etc). The one exception is the UNIX file descriptor for the dag card (returned by `dag_open`), this is a standard `int` type, to maintain consistency with existing Endace API's.

Usage Example

The following is a typical example of how the EMA library is used. It is assumed that there is a DAG 3.7T card installed at location 0. Error checking has been removed for brevity.

```
#include <dagapi.h>
#include <dagema.h>
#define MSG_ID 0x12345678
#define MSG_LENGTH 1024
...

void example_function (void)

{
char dagname[DAGNAME_BUFSIZE];
int stream;
int dagfd;
uint8_t send_msg[MSG_LENGTH];
uint8_t recv_msg[MSG_LENGTH];
uint32_t msg_len;
uint32_t msg_id;
/* open the dag card at location 0 */

dag_parse_name (“dag0”, dagname, DAGNAME_BUFSIZE, &stream);

dagfd = dag_open (dagname);
/* reset the embedded processor card to put in a known state */
/* run a memory test during reset */
dagema_reset_processor (dagfd, EMA_RUN_DRAM_MEMORY_TEST);

/* open a connection to the processor */
dagema_open_conn (dagfd);
...
/* send and receive messages */

dagema_send_msg (dagfd, MSG_ID, MSG_LENGTH, send_msg, NULL);
msg_len = MSG_LENGTH;
dagema_recv_msg (dagfd, &msg_id, &msg_len, recv_msg, NULL);

...

/* close the connection */
dagema_close_conn (dagfd, 0);

/* cleanup */
dag_close (dagfd);
}
```

 ©2006-2007 Endace Technology Ltd. Confidential - Version 2: October 2007

dagema_le16toh Function

Purpose Converts a 16 bit number from little endian to host byte order.

Declared In `dagema.h`

Prototype `uint16_t dagema_le16toh (uint16_t little16)`

Parameters > `little16`

Little endian number to convert to host byte order

Returns Returns the host byte ordered value

Comments On little endian host processors (for example Intel processors) this function returns the little endian argument unchanged. These functions, along with the other endian conversion functions, were implemented to provide a complete API across both little endian and big endian processors.

Note: By convention data structures and their fields sent in message payloads are in little endian order, however this is not enforced, and the user is free to implement any endian order required by the embedded software.

dagema_le32toh Function

Purpose Converts a 32 bit number from little endian to host byte order.

Declared In `dagema.h`

Prototype `uint32_t dagema_le32toh (uint32_t little32)`

Parameters > `little32`

Little endian number to convert to host byte order

Returns Returns the host byte ordered value

Comments On little endian host processors (for example Intel processors) this function returns the little endian argument unchanged.

dagema_htole16 Function

Purpose Converts a 16 bit number from host to little endian byte order.

Declared In `dagema.h`

Prototype `uint16_t dagema_htole16 (uint32_t host16)`

Parameters > `host16`

Host byte ordered number to convert to little endian byte order.

Returns Returns the little endian byte ordered value.

Comments On little endian host processors (for example Intel processors) this function returns the little endian argument unchanged.

dagema_htole32 Function

Purpose Converts a 32 bit number from host to little endian byte order.

Declared In `dagema.h`

Prototype `uint32_t dagema_htole32 (uint32_t host 32)`

Parameters > `host32`

host byte ordered number to convert to little endian byte order.

Returns Returns the host byte ordered value

Comments On little endian host processors (for example Intel processors) this function returns the little endian argument unchanged.

dagema_get_last_error Function

Purpose Returns the last error code generated by the dagema library call.

Declared In `dagema.h`

Prototype `uint32_t dagema_get_last_error (void)`

Parameters none

Returns Returns thelast error code generated by an unsuccessful call to one of the dagema functions.

Comments Every dagema function clears the last error value before attempting to do anything, this means that an error code generated by a previous call to a dagema function will be reset to ENONE when a new dagema function is called.

The last error code is local to the process that is using the dagema library, if multimple processes are using the dagema library, separate last error values are used. Possible error codes are documented in the dagema functions that generate the errors.

 ©2006-2007 Endace Technology Ltd. Confidential - Version 2: October 2007

dagema_open_conn Function

Purpose Opens a connection to the specified card.

Declared In `dagema.h`

Prototype `int dagema_open_conn (int dagfd)`

Parameters > `dagfd`

DAG descriptor provided by `dag_open()`

Returns Returns `0` if the connection was opened, otherwise `-1` is returned. Call `dagema_get_last_error()` to retrieve the error code.

Possible error codes:

EBADF (bad file descriptor)

EEXIST (a connection has already been opened to this card)

ECARDNOTSUPPORTED (card type not supported)

ENOMEM (out of memory)

ELOCKED (another process already has an open connection)

ERESP (did not receive a response or received an invalid response from the card)

Comments This function must be called to open a connection before attempting to send or receive any messages. The DAG descriptor must remain valid while a connection is opened, for example:

```
/* The following is invalid and will cause runtime
errors */
dagfd = dag_open ("/dev/dag0");
dagema_open_conn (dagfd);
dag_close (dagfd);

....   /* here the dagema library expects the DAG */
/* descriptor to be valid */

dagema_close_conn (dagfd, 0);
```

Instead `dagema_close_conn` should be called before the DAG descriptor is disposed of.

Only one connection to a card can be opened at a time, across all processes. However a process is not limited to a connection to a single card, each process can open a connection to up to 16

If a connection is not opened successfully, try resetting the card using `dagema_reset_processor` to put it in a known state, then retry opening a connection.

Only DAG 3.7T and DAG 7.1S cards can be used with this function, attempting to open a connection to another card will result in a ECARDNOTSUPPORTED error.

All open connections should be closed using `dagema_close_conn` prior to the process terminating, failure to do so will cause unpredictable behavior.

If this function returns `1` with an error code of EEXISTS, then the connection has already been open, messages can safely be sent and received.

dagema_close_conn Function

Purpose Closes a connection to the specified card.

Declared In `dagema.h`

Prototype `int dagema_close_conn (int dagfd, unit32_t flags)`

Parameters > `dagfd` DAG descriptor provided by `dag_open`, this should be the same value provided to `dagema_open_conn`.

> `flags`

Optional close flags, multiple flags can be OR'ed together. See the comments below for possible values.

Returns Returns 0 if the connection was opened, otherwise `1` is returned. Call `dagema_get_last_error()` to retrieve the error code.

Possible error codes:

EBADF (bad file descriptor)

Comments The close function terminates the connection to the card. This function flushes the internal messaging buffers to the card prior to returning, therefore depending on the number of messages queued, it may take some time to complete.

Possible values that can supplied in the flags argument are (multiple can be OR'ed together)

EMA_CLOSE_NO_FLUSH

Closes the connection, without waiting for queued messages to be sent and without waiting for partial messages to be received from the card. If this flag

is used it is recommended that the card be reset (using `dagema_reset_processor)` prior to opening another connection.

 ©2006-2007 Endace Technology Ltd. Confidential - Version 2: October 2007

dagema_reset_processor Function

Purpose Resets the embedded processor on the card.

Declared In `dagema.h`

Prototype `int dagema_reset_processor (int dagfd, unit32_t flags)`

Parameters

> `dagfd`

DAG descriptor provided by `dag_open`.

> `flags`

Optional close flags, multiple flags can be OR'ed together. See the comments below for possible values.

Returns Returns `0` if the connection was opened, otherwise `1` is returned. Call `dagema_get_last_error()` to retrieve the error code.

Possible error codes:

`EBADF` (bad file descriptor)

`EEXIST` (there is currently a connection opened to this card)

`ELOCKED` (another process has an open connection to the card)

`ECARDNOTSUPPORTED` (the card type is not supported)

`ETIMEDOUT` (timed-out waiting for the reset process to complete)

`ECARDMEMERROR` (there was a problem detected with the memory used for the embedded network processor)

Comments To reset the embedded processor on a card, no application can have an open connection to the card, if a connection is open `1` will be returned and `dagem_get_last_error` will return either EEXIST or ELOCKED.

Important: The reset process may take up to two minutes to complete (especially if memory tests are performed), during this process the function blocks, therefore it may appear that the function is stalled. As an alternative consider using `dagema_reset_processor_with_cb` instead, this version performs the same reset process, but calls a callback function with status updates during the reset process.

Possible values for the flags argument (multiple can be OR'ed together):

• EMA_RUN_DRAM_MEMORY_TEST

Valid for both DAG 3.7T and DAG 7.1S cards. It tells the card to run DRAM memory tests as part of a reset. The memory tested is internal to the card and used for the embedded processor, this does not test the memory on the host. If the memory tests reveal a problem with the memory, `1` is returned and `dagema_get_last_error` will return ECARDMEMERROR.

• EMA_RUN_CPP_DRAM_MEMORY_TEST

Valid for the DAG 7.1S card only, this flag is ignored if specified for a DAG 3.7T card. It tells the card to run microengine memory tests as part of a reset. If the memory tests reveal a problem, `-1` is returned and `dagema_get_last_error` will return ECARDMEMERROR.

©2006-2007 Endace Technology Ltd. Confidential - Version 2: October 2007

dagema_reset_processor_with_cb Function

Purpose Resets the embedded processor on the card, with a callback function to display status.

Declared In `dagema.h`

Prototype

```
int dagema_reset_processor_cb (int dagfd, unit32_t
flags, reset_handler_t cb)
```

Parameters `dagfd`

DAG descriptor provided by `dag_open`

> `flags`

Optional reset flags, multiple can OR'ed together, see the comments in section 3.8 for possible values.

> `cb` Pointer to a callback function that will be called with status information during the reset process.

Returns Returns `0` if the reset process complete successfully, otherwise `1` is returned. Call `dagema_get_last_error()` to retrieve the error code.

Possible error codes:

EBADF(bad file descriptor)

EEXIST (there is currently a connection opened to this card)

ELOCKED (another process has an open connection to the card)

ECARDNOTSUPPORTED (the card type is not supported)

ETIMEDOUT (timed-out waiting for the reset process to complete)

ECARDMEMERROR (there was a problem detected with the memory used for the embedded processor)

Comments This routine is functionally equivalent to `dagema_reset_processor` except that it calls a callback function during the reset process. The callback function should have the following prototype:

```
int reset_callback(uint32_t stage);
```

If the callback function returns 0 the reset process continues, if any other value is returned the reset process terminates with a 1 return value and copies the value returned by the callback function into the last error variable (can be retrieved using `dagema_get_last_error`). The stage argument passed to the callback function, can have one of the following values (some values may be skipped depending on the processor in use and the flags set):

EMA_RST_INIT Always the first stage called, this occurs before the reset process has started.

EMA_RST_BOOTLOADER_STARTED Indicates the bootloader has started.

EMA_RST_MEMORY_INIT Indicates the memory controller on the embedded processor has been initialized. This stage occurs twice on DAG 7.1S cards, once for the XSI memory controller and once for the CPP memory controller.

EMA_RST_STARTING_MEM_TEST This stage occurs just before the memory tests are started. This only occurs if EMA_RUN_DRAM_MEMORY_TEST or EMA_RUN_CPP_DRAM_MEMORY_TEST is set in the flags. On DAG 7.1S cards this stage may occur twice, once for each memory test. Depending on the hardware a memory test may take up to 1 minute to complete.

EMA_RST_FINISHED_MEM_TEST Indicates the memory test has completed successfully.

EMA_RST_KERNEL_BOOTED Indicates the software on the embedded processor has started.

EMA_RST_DRIVER_STARTED Indicates the driver for the embedded side of the messaging has started. This stage is only available on the DAG 7.1S, other cards don't indicate when the embedded driver is running.

EMA_RST_COMPLETE Final stage called just before the reset process completes.

 ©2006-2007 Endace Technology Ltd. Confidential - Version 2: October 2007

dagema_send_msg Function

Purpose Sends a message to the embedded processor.

Declared In `dagema.h`

Prototype
```
int dagema_send_msg  (int dagfd, unit32_t message_id,
uint32_t length, const uint8_t *tx_message, uint8_t
*trans_id))
```

Parameters
> `dagfd` DAG descriptor provided by `dag_open`, this should be the same value provided to `dagema_open_conn`.

> `message_id` The 32-bit id of the message to send.

> `length` The length of the message payload in bytes to send. Message payloads must not exceed 2048 bytes, if a larger length is supplied the call will fail. This argument may be 0 in which case `tx_message` is ignored and a message with a zero byte payload is sent.

> `tx_message` Buffer containing the message payload to send.

< `trans_id` Pointer to an 8-bit value that receives the transaction Id of the message sent. By convention the embedded processor will use the same transaction id for response messages. This can be used to determine which response corresponds to which request. This argument is optional, set it to NULL if not required.

Warning: This feature is not fully implemented on all cards, if zero is returned in `trans_id` it indicates the transaction id should not be used.

Returns Returns 0 if the message was sent successfully, otherwise 1 is returned. Call `dagema_get_last_error()` to retrieve the error code.

Possible error codes:

EBADF (bad file descriptor)

ERANGE (length was greater than the maximum length allowed, 2048 bytes)

EINVAL (invalid arguments)

Error codes for send(2)

Comments Messages are internally queued by this library, therefore the exact time the message is sent to the card is determinant on the amount of messages already in the queue and the speed of the embedded processor. However a copy of the `tx_message` buffer is stored internally in the library, so it can be freed/changed when this function returns.

This function returns when the messages has been added to the internal transmit queue, therefore a positive return value is not necessarily an indication that the message was sent to the card correctly.

dagema_recv_msg Function

Purpose Reads a message from the embedded processor.

Declared In `dagema.h`

Prototype

```
int dagema_recv_msg (int dagfd, uint32_t *message_id,
uint32_t *length, const uint8_t *rx_message, uint8_t
*trans_id)
```

Parameters

> `dagfd` DAG descriptor provided by `dag_open`, this should be the same value provided to `dagema_open_conn`.

< `message_id` Pointer to a 32-bit value that will receive the id of the received message.

< > length Upon entry this argument should contain the maximum size of the `rx_message` buffer in bytes. Upon exit it will contain the size of the received messages payload, this may be larger than the `rx_message` buffer size.

< `rx_message` Buffer that will receive the message payload.

< `trans_id` Pointer to an 8-bit value that receives the transaction Id of the message received.

Warning: This feature is not fully implemented on all cards, if zero is returned in `trans_id` it indicates the transaction id should not be used.

Returns Returns 0 if a message was received successfully, otherwise 1 is returned. Call `dagema_get_last_error()` to retrieve the error

code.

Possible error codes:

EBADF (bad file descriptor)

EINAL (invalid arguments)

ERESP (corrupt message received)

Error codes for recv(2)

Comments This function blocks until either a message arrives or an error occurs.

If the `rx_message` buffer supplied is not big enough to accommodate the complete received message, the `rx_message` buffer is filled with data, the overflow is discarded. Upon exit the length will contain the actual size of the message payload, not the amount of data copied into the `rx_message` buffer. For the above reason it is recommended to supply a buffer that can contain the maximum size of the message, 2048 bytes.

This function shouldn't be called if a message handler has been installed (using `dagema_set_msg_handler`), otherwise this function will block indefinitely

©2006-2007 Endace Technology Ltd. Confidential - Version 2: October 2007

dagema_recv_msg_timeout Function

Purpose Reads a message from the embedded processor, with a timeout.

Declared In `dagema.h`

Prototype

```
int dagema_recv_msg_timeout (int dagfd, uint32_t
*message_id, uint32_t *length, const uint8_t
*rx_message, uint8_t *trans_id, uint32_t timeout)
```

Parameters

> `dagfd` DAG descriptor provided by `dag_open`, this should be the same value provided to `dagema_open_conn`.

< `message_id` Pointer to a 32-bit value that will receive the id of the received message.

< > `length` Upon entry this argument should contain the maximum size of the `rx_message` buffer in bytes. Upon exit it will contain the size of the received messages payload which may be larger than the `rx_message` buffer size.

< `rx_message` Buffer that will receive the message payload.

< `trans_id` Pointer to an 8-bit value that receives the transaction Id of the message received.

Warning: This feature is not fully implemented on all cards, if zero is returned in `trans_id` it indicates the transaction id should not be used.

> `timeout` The number of milliseconds to wait for a message before returning.

Returns Returns `0` if a message were received successfully, otherwise `1` is returned. Call `dagema_get_last_error()` to retrieve the error code.

Possible error codes:

EBADF (bad file descriptor)

EINAL (invalid arguments)

ERESP (corrupt message received)

ETIMEDOUT (timed out waiting for a message)

Error codes for `recv(2)`

Comments This function waits the prescribed amount of milliseconds for a message to arrive, if no message is received `1` is returned and ETIMEDOUT is set as the last error code.

Besides the addition of a time-out, this function is identical to `dagema_recv_message`.

©2006-2007 Endace Technology Ltd. Confidential - Version 2: October 2007

dagema_set_msg_handler Function

Purpose Installs a callback handler that is called when a message arrives.

Declared In `dagema.h`

Prototype `int dagema_set_msg_handler (int dagfd, msg_handler_t msg_handler)`

Parameters > `dagfd` DAG descriptor provided by `dag_open`, this should be the same value provided to `dagema_open_conn`.

> `msg_handler` Callback function to call when a message arrives.

Returns Returns 0 if message handler was installed successfully, otherwise `1` is returned. Call `dagema_get_last_error()` to retrieve the error code.

Possible error codes:

- EBADF (bad file descriptor)

Comments The callback function should have the following prototype;

```
void msg_handler(uint32_t message_id, uint8_t
*rx_message, uint32_t length, uint8_t trans_id);
```

When a message is received the library will check its integrity and then pass the details onto the callback function.

The callback function may be called from a different thread than the one that installed the message handler. It is the callers responsibility to ensure all resources shared between a callback function and the main process are thread safe.

This function can be called multiple times, with each successive call replacing the previous callback handler. If you want to remove an existing callback handler pass in NULL for the `msg_handler` argument.

 ©2006-2007 Endace Technology Ltd. Confidential - Version 2: October 2007

Version History

Version	Date	Reason
1	January 2006	Initial Version
2	October 2007	New template