GT.M Call-Ins on UNIX: Invoking M from C

Technical Bulletin 5-027

11 July 2003

THE GREYSTONE GROUP

SANCHEZ
1. IDENTIFICATION

   Release:   GT.M Version 4.4-002 and later

   Issued By:  Greystone Group
               Sanchez Computer Associates, Inc
               40 Valley Stream Parkway
               Malvern, PA 19355 USA

   GT.M Support:  +1 (610) 578-4226
   Switchboard:  +1 (610) 295-8877
   FAX:          +1 (610) 296-7371
   Web:          http://www.sanchez-gtm.com
   E-mail:       gtmsupport@sanchez.com

   Issue Date:  July 11, 2003

2. SYSTEM ENVIRONMENTS

   This functionality is available in GT.M 4.4-002 release, and onwards and does not require any special or separate installation.

3. INTRODUCTION

   Call-In is a framework supported by GT.M that allows a C/C++ program to invoke an M routine within the same process context. GT.M provides a well-defined call-in interface packaged as a run-time shared library that can be linked into an external C/C++ program.

   This document specifies the GT.M call-in facility for UNIX platforms.

4. RELEVANT FILES

   To facilitate call-ins to M routines, the GT.M distribution directory ($gtm_dist) contains the following files:

   1. libgtmshr.so - A shared library that implements the GT.M run-time system, including the call-in API. If call-ins are used from a standalone C/C++ program, this library needs to be explicitly linked into the program. Section 6
"Building Standalone Programs" describes the necessary linker options on each supported platform.

NOTE: so is the recognized shared library file extension on most UNIX platforms, except on HP-UX (.sl).

2. mumps - The GT.M startup program that dynamically links with libgtmshr.so.

3. gtmxc_types.h - A C header file containing the declarations of call-in API.

The following sections describe the files relevant to using call-ins.

4.1 gtmxc_types.h

The header file provides signatures of all call-in interface functions and definitions of those valid data types that can be passed from C to M. Sanchez strongly recommends that these types be used instead of native types (int, char, float etc.), to avoid possible mismatch problems during parameter passing.

gtmcs_types.h defines the following types that can be used in call-ins.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Void</td>
<td>No return value – a simple M routine call</td>
</tr>
<tr>
<td>gtm_long_t</td>
<td>32-bit signed integer value</td>
</tr>
<tr>
<td>gtm_ulong_t</td>
<td>32-bit unsigned integer value</td>
</tr>
<tr>
<td>gtm_float_t</td>
<td>Single precision floating point value supported by the platform</td>
</tr>
<tr>
<td>gtm_double_t</td>
<td>Double precision floating point value supported by the platform</td>
</tr>
<tr>
<td>gtm_long_t*</td>
<td>Pointer to 32-bit signed integer value</td>
</tr>
<tr>
<td>gtm_ulong_t*</td>
<td>Pointer to 32-bit unsigned integer value</td>
</tr>
<tr>
<td>gtm_float_t*</td>
<td>Pointer to single precision floating point value</td>
</tr>
<tr>
<td>gtm_double_t*</td>
<td>Pointer to double precision floating point value</td>
</tr>
<tr>
<td>gtm_char_t*</td>
<td>Pointer to null-terminated string</td>
</tr>
<tr>
<td>gtm_string_t*</td>
<td>Pointer to a structure described below, useful for bulk transfer</td>
</tr>
</tbody>
</table>
### 4.2 Call-In Table

The call-in table file is a text file that contains the signatures of all M label references that get called from C. In order to pass the typed C arguments to the type-less M formallist, the environment variable `GTMCI` must be defined to point to the call-in table file path. Each signature must be specified separately in a single line. GT.M reads this file and interprets each line according to the following convention (specifications within box brackets `[]`, are optional):

\[
<\text{c-call-name}> : <\text{ret-type}> <\text{label-ref}> ([<\text{direction}>:<\text{param-type}>,…])
\]

Where,

- `<label-ref>` is the entry point (i.e. a valid label reference) at which GT.M starts executing the M routine being called-in.
- `<c-call-name>` is a unique C identifier that is actually used within C to refer to `<label-ref>` (see Section on “Call an M Routine from C” later in this document).
- `<direction>` is either `I` (input-only), `O` (output-only), or `IO` (input-output).
- `<ret-type>` is the return type of `<label-ref>`.

**NOTE:** Since the return type is considered as an output-only (O) parameter, the only types allowed are pointer types and `void`. `Void` cannot be specified as parameter.

- `<param-type>` is a valid parameter type (see Table 1. `gtmxc_types.h` types). Empty parentheses should be specified if no argument is passed to `<label-ref>`.

---

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gtm_status_t</code></td>
<td>A 32-bit status word indicating the returning status from GT.M.</td>
</tr>
</tbody>
</table>

typedef struct {
    gtm_long_t   length;
    gtm_char_t*  address;
} gtm_string_t;

The pointer types defined above are always 32 bit addresses on all platforms, even on Tru64 UNIX that supports 64 bit addresses. These types must be used on Tru64 UNIX instead of the native types. Refer to the section on “HP Alpha/AXP Tru64 UNIX” later in this document.
The `<direction>` indicates the type of operation that GT.M performs on the parameter read-only (I), write-only (O), or read-write (IO). All O and IO parameters must be passed by reference, that is as pointers since GT.M writes to these locations. All pointers that are being passed to GT.M must be pre-allocated. The following table details valid type specifications for each `direction`.

<table>
<thead>
<tr>
<th>Directions</th>
<th>Allowed parameter types</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td><code>gtm_long_t, gtm_ulong_t, gtm_float_t, gtm_double_t, gtm_long_t*, gtm_ulong_t*, gtm_float_t*, gtm_double_t*, gtm_char_t*, gtm_string_t*</code></td>
</tr>
<tr>
<td>O/IO</td>
<td><code>gtm_long_t*, gtm_ulong_t*, gtm_float_t*, gtm_double_t*, gtm_char_t*, gtm_string_t*</code></td>
</tr>
</tbody>
</table>

An example of call-in table (calltab.ci) for piece.m (refer to the section "Example – piece.m" later in this document):

```c
print : void display^piece()

gtpiece : gtm_char_t* get^piece(I:gtm_char_t*, I:gtm_char_t*, I:gtm_long_t)

setpiece : void set^piece(IO:gtm_char_t*, I:gtm_char_t*, I:gtm_long_t, I:gtm_char_t*)

pow : gtm_double_t* pow^piece(I:gtm_double_t, I:gtm_long_t)

powequal : void powequal^piece(IO:gtm_double_t*, I:gtm_long_t)

piece : gtm_double_t* pow^piece(I:gtm_double_t, I:gtm_long_t)
```

*Note: The same entryref can be called by different C call names (for example, `pow` and `piece`). However, if there are multiple lines with the same call name, only the first entry will be used by GT.M. GT.M ignores all subsequent entries using a call name. Also, note that the second and third entries, although shown here as wrapped across lines, must be specified as a single line in the file.*
5. CALL-IN INTERFACE

5.1 Initialize GT.M

```c
#include <gtm.h>

int gtm_init(void);
```

If the base program is not an M routine but a standalone C program, `gtm_init()` must be called (before calling any GT.M functions), to initialize the GT.M run-time system.

`gtm_init()` returns zero (0) on success. On failure, it returns the GT.M error status code whose message can be read into a buffer by immediately calling `gtm_zstatus()` (refer to the section "Print Error Messages" later in this document). Duplicate invocations of `gtm_init()` are ignored by GT.M.

If call-ins are used from an external call function (that is, a C function that has itself been called from M code), `gtm_init()` is not needed, because GT.M is initialized before the External Call. All `gtm_init()` calls from External Calls functions are ignored by GT.M.

5.2 Call an M Routine from C

```c
#include <gtm.h>

int gtm_ci(const gtm_char_t* c_call_name, ...
```

The variable argument function `gtm_ci()` is the interface that actually invokes the specified M routine and returns the results via parameters. The `gtm_ci()` call must obey the following format:

```c
status = gtm_ci(c_call_name [, ret_val] [, arg1] ...);
```

- First argument - `c_call_name`, a null-terminated C character string indicating the alias name for the corresponding `<lab-ref>` entry in the call-in table.

- Optional second argument – `ret_val`, a pre-allocated pointer through which GT.M returns the value of QUIT argument from the (extrinsic) M routine. `ret_val` must be the same type as specified for `<ret-type>` in the call-in table entry. The `ret_val` argument is needed if and only if `<ret-type>` is not `void`.

- Optional list of arguments to be passed to the M routine’s formalist - the number of arguments and the type of each argument must match the number of parameters and parameter types specified in the corresponding call-in table entry. All pointer arguments must be pre-allocated. GT.M assumes that any pointer, which is passed for O/IO-parameter points to valid write-able memory.
The status value returned by `gtm_ci()` indicates the GT.M status code; zero (0), if successful, or a non-zero; $ZSTATUS error code on failure. The $ZSTATUS message of the failure can be read into a buffer by immediately calling `gtm_zstatus()` (refer to the section "Print Error Messages" later in this document).

Examples:
```c
#include "gtmxc_types.h"

gtm_status_t status;
gtm_char_t piece[100];
gtm_double_t mant, powval;
gtm_long_t exp;

status = gtm_ci("getpiece", &piece[0], "GT.M call-ins", " ", 1);
status = gtm_ci("pow", &powval, mant, exp);
status = gtm_ci("powequal", &powval, exp);
```

5.3 Print Error Messages

```c
void gtm_zstatus (gtm_char_t* msg_buffer, gtm_long_t buf_len);
```

This function returns the null-terminated $ZSTATUS message of the last failure via the buffer pointed by `msg_buffer` of size `buf_len`. The message is truncated to size `buf_len` if it does not fit into the buffer. `gtm_zstatus()` is useful if the external application needs the text message corresponding to the last GT.M failure. A buffer of 2048 is sufficient to fit in any GT.M message.

5.4 Exit from GT.M

```c
gtm_status_t gtm_exit (void);
```

`gtm_exit()` can be used to shut down all databases and exit from the GT.M environment that was created by a previous `gtm_init()`.

Note that `gtm_init()` creates various GT.M resources and keeps them open across multiple invocations of `gtm_ci()` until `gtm_exit()` is called to close all such resources. On successful exit, `gtm_exit()` returns zero (0), else it returns the $ZSTATUS error code.

`gtm_exit()` cannot be called from an external call function. GT.M reports the error `GTM-E-INVGTMEXIT` if an external call function invokes `gtm_exit()`. Since the GT.M run-time system must be operational even after the external call function returns, `gtm_exit()` is meant to be called only once during a process lifetime, and only from the base C/C++ program when GT.M functions are no longer required by the program.
5.5 Callback Functions

gtm_hiber_start(), gtm_hiber_start_wait_any(), gtm_start_timer(),
gtm_cancel_timer(), gtm_malloc() and gtm_free(). These functions, which have
been available through the existing external call mechanism, are available as
exported symbols from the shared library libgtmshr.so and can be called from C as
well. Refer to the GT.M Programmer’s Guide (V4.3 and later) for details on using the
callbacks.

5.6 Example – “piece.m”

The following M routine (piece.m)

display ;
    write "hello, world",!
    quit
get(str,delim,start)  ;
    quit $piece(str,delim,start)
set(str,delim,start,value);
    set $piece(str,delim,start)=value
    quit
pow(num,exp)  ;
    quit num**exp
powequal(num,exp);
    set num=num**exp
    quit

can be called from the following C program (main.c)
#include <stdio.h>
#include "gtmxc_types.h"
#define BUF_LEN 1024
int main()
{

gtm_char_t msgbuf[BUF_LEN], dststr[100];
gtm_char_t srcstr[] = "/usr/library/V43001E";
gtm_status_t status;
status = gtm_init();
if (status != 0)
{
by creating the call-in table (refer to the section "Call-in Table" earlier in this document) in the current directory with the following Bourne shell commands (in Linux):

```
$ cc -c main.c -I$gtm_dist  # compile
$ ld main.o -o myprog -L$gtm_dist -rpath $gtm_dist -lgtmshr -lc
```
$export GTMCI=`pwd`/calltab.ci  # call-in table for GT.M
$. ./myprog  # run application

The above example produces the following output:
srcstr = /usr/library/V43001E
$piece(/usr/library/V43001E,"/",4) is V43001E
After set $piece(srcstr,"/",4) = V44002
srcstr = /usr/library/V44002

NOTE: The ld –rpath option above records $gtm_dist path in the executable so that the shared library libgtmshr.so can be found at runtime. The name of this option varies on different platform. The cc and ld options may also vary depending on the platform. Refer to the appropriate system’s manual for details on each platform.

6. BUILDING STANDALONE PROGRAMS

All external C functions that use call-ins should include the header file gtmxc_types.h that defines various types and provides signatures of call-in functions. To avoid potential size mismatches with the parameter types, Sanchez strongly recommends that gtm *t types defined in gtmxc_types.h be used instead of the native types (int, float, char, etc).

To use call-ins from a standalone C program, it is necessary that the GT.M runtime library (libgtmshr.so) is explicitly linked into the program. If call-ins are used from an External Call function (which in turn was called from GT.M through the existing external call mechanism), the External Call library does not need to be linked explicitly with libgtmshr.so since GT.M would have already loaded it.
The following sections describe compiler and linker options that must be used on each platform for call-ins to work from a standalone C/C++ program. If call-ins are used from external call libraries, refer to GT.M Programmer’s Guide (V4.3 or later) to build external call shared libraries.

6.1 IBM pSeries (RS/6000) AIX
- Compiler: -I$gtm_dist
- Linker: -L$gtm_dist -lgtmshr

6.2 HP Alpha/AXP Tru64 UNIX
- Compiler: -xtaso -xtaso_short -I$gtm_dist
• Linker: -taso -L$gtm_dist -lgtmshr -rpath $gtm_dist

On Tru64, it is absolutely required that the program be built with short addressing (taso) options since libgtmshr.so is built to work within 32-bit process address space. GT.M does not work without taso options.

6.3 HP Series 9000 HP-UX

• Compiler: -I$gtm_dist
  • Linker: -L$gtm_dist -lgtmshr +b $gtm_dist

6.4 Sun SPARC Solaris

• Compiler: -I$gtm_dist
  • Linker: -L$gtm_dist -lgtmshr -R $gtm_dist

6.5 X86 GNU/Linux

• Compiler: -I$gtm_dist
  • Linker: -L$gtm_dist -lgtmshr -rpath $gtm_dist

NOTE: Sanchez advises that the C/C++ compiler front-end be used as the Linker to avoid specifying the system startup routines on the ld command. The compile can pass linker options to ld using –W option (for example, cc –W1,-R,$gtm_dist). For more details on these options, refer to the appropriate system’s manual on the respective platforms.

7. NESTED CALL-INS

Call-ins can be nested by making an external call function in-turn call back into GT.M. Each gtm_ci() called from an External Call library creates a call-in base frame at $ZLEVEL 1 and executes the M routine at $ZLEVEL 2. The nested call-in stack unwinds automatically when the External Call function returns to GT.M.

GT.M currently allows up to 10 levels of nesting, if TP is not used, and less than 10 if GT.M supports call-ins from a transaction (refer to the “Limitations” section in this document). GT.M reports the error GTM-E-CIMAXLEVELS when the nesting reaches its limit.
Following are the GT.M commands, Intrinsic Special Variables, and functions whose behavior changes in the context of every new nested call-in environment.

**ZGOTO** operates only within the current nested M stack. ZGOTO zero (0) unwinds all frames in the current nested call-in M stack (including the call-in base frame) and returns to C. ZGOTO one (1) unwinds all current stack frame levels up to (but not inclusive) the call-in base frame and returns to C, while keeping the current nested call-in environment active for any following `gtm_ci()` calls.

**$ZTRAP/$ETRAP** NEW’d at level 1 (in GTM$CI frame).

**$ZLEVEL** initializes to one (1) in GTM$CI frame, and increments for every new stack level.

**$STACK** initializes to zero (0) in GTM$CI frame, and increments for every new stack level.

**$ESTACK** NEW’d at level one (1) in GTM$CI frame.

**$ECODE/$STACK()** initialized to null at level one (1) in GTM$CI frame.

*NOTE: After a nested call-in environment exits and the external call C function returns to M, the above ISVs and Functions restore their old values.*

### 8. ERROR MESSAGES

**GTM-E-INVGMTMEXIT**, Inappropriate invocation of gtm_exit. gtm_exit cannot be invoked from external calls.

This run-time error is reported as a result of invalid usage of `gtm_exit()` from an external call function. `gtm_exit()` must not be called from external call functions.

**GTM-E-CITPNESTED** Call-ins cannot be used inside a TP transaction

A run-time error reported when an external call function, called from inside TSTART/TCOMMIT fence, invokes `gtm_ci()`. Call-ins are not currently allowed from inside TP.

**GTM-E-CIMAXLEVELS** Too many nested Call-ins. Nested resources exhausted at level xxxx.

A run-time error reported when GT.M runs out of its internal condition handlers stack. Ensure that call-ins are not nested in uncontrolled manner.
GTM-E-CIMAXPARAM Exceeded maximum number of parameters in the call-in table entry. An M routine cannot accept more than 32 parameters.

A run-time call-in table parses the error reported when the call-in table specified by $GTMCII contains more than 32 parameters. Since an M formal list can only accept up-to 32 parameters, user cannot pass more than 32 arguments to gtm_ci(), excluding <c-call-name> and <ret-type>.

9. LIMITATIONS

1. External calls must not be fenced with TSTART/TCOMMIT if the external routine calls back into mumps using call-in mechanism. GT.M reports the error GTM-E-CITPNESTED if nested call-ins are invoked within a TP fence since GT.M currently does not handle TP support across multiple call-in invocations.

2. The external application should never call exit() unless it has called gtm_exit() previously. GT.M internally installs an exit handler that should never be bypassed to have a clean shutdown of GT.M.

3. The external application should never use any signals when GT.M is active since GT.M reserves them for its internal use. GT.M provides the ability to handle SIGUSR1 within M (refer to $ZINTERRUPT section in Intrinsic Special Variables chapter of GT.M Programmer’s Guide for more information). An interface is provided by GT.M for timers. Although not required, Sanchez recommends the use of gtm_malloc() and gtm_free() for memory management by C code that executes in a GT.M process space for enhanced performance and improved debugging.

10. CONVERSION NOTES

All GT.M callback functions (timers and gtm_malloc/gtm_free) are now exported as public symbols from the shared library libgtmshr.so.

NOTE: Sanchez strongly recommends that the existing mechanism be replaced with the direct calls to these functions. Although the old mechanism is still supported, Sanchez recommends the upgrade for better maintenance of external call code. The old mechanism may be deprecated in future.