**Thread Local Storage Specification for C6000 EABI**

**Revision History**

| Version | Primary Author(s) | Description of Version | Date Completed |
| --- | --- | --- | --- |
| v0.1 | Anbu Eashver | Initial Version | 1/9/2012 |
| *v.02* | *Anbu Eashver* | *Updated the static exe TLS initialization based on feedback* | *1/17/2012* |
| *v.03* | *Anbu Eashver* | *Added far-DP GOT addressing* | *1/24/2012* |
| *v.03* | *Anbu Eashver* | *Minor edits to remove inconsistencies* | *2/22/2012* |
|  |  |  |  |

**Table of Contents**

[1 Overview 3](#_Toc317779674)

[2 Terms and Concepts 3](#_Toc317779675)

[3 User Interface 4](#_Toc317779676)

[4 ELF Object File Representation 4](#_Toc317779677)

[5 TLS Access Models 5](#_Toc317779678)

[5.1 C6x Linux TLS models 5](#_Toc317779679)

[5.1.1 General Dynamic TLS Access Model 6](#_Toc317779680)

[5.1.2 Local Dynamic TLS Access Model 8](#_Toc317779681)

[5.1.3 Initial Exec TLS Access Model 9](#_Toc317779682)

[5.1.4 Local Exec TLS Model 10](#_Toc317779683)

[5.2 Static Executable TLS model 10](#_Toc317779684)

[5.2.1 Static Executable Addressing 11](#_Toc317779685)

[5.2.2 Static exe TLS Runtime Architecture 11](#_Toc317779686)

[5.2.3 Static Exe TLS allocation 12](#_Toc317779687)

[5.2.4 Main Thread’s TLS Initialization 13](#_Toc317779688)

[5.2.5 TLS Initialization by Thread Library 13](#_Toc317779689)

[5.2.6 Thread Pointer 13](#_Toc317779690)

[5.3 Bare-Metal Dynamic Linking TLS Model 13](#_Toc317779691)

[5.3.1 Default TLS Addressing for Bare-Metal Dynamic Linking 14](#_Toc317779692)

[5.3.2 TLS block creation 15](#_Toc317779693)

[6 Thread-Local Symbol Resolution 15](#_Toc317779694)

[6.1 General/Local Dynamic TLS Weak Reference Addressing 15](#_Toc317779695)

[6.2 Initial and Local Exec TLS Weak Reference Addressing 16](#_Toc317779696)

[6.3 Static TLS Model Weak References 16](#_Toc317779697)

[7 Thread-Local Relocations 16](#_Toc317779698)

[8 References 20](#_Toc317779699)

**Table of Figures**

[Figure 5.1 C6x Linux TLS Run-Time Representation 5](#_Toc315395146)

[Figure 5.2. Static Executable TLS Run-Time Representation 11](#_Toc315395147)

[Figure 5.4. Bare-Metal Default TLS Run-Time Representation 15](#_Toc315395148)

# Overview

Use of multi-threading programming is common in many embedded systems using the C6000 family of processors from Texas Instruments. Given the increase in the number of C6000 CPU based multi-core devices, multi-threading programming is expected to be even more widely adopted to leverage the multiple cores. Also, multi-core programming models like OpenMP and OpenCL normally rely on underlying multi-threading support.

These complex multi-threading programs can be structured better and easier to develop if the threads can use variables that have static storage duration and specific to the thread. That is, other threads cannot see or access this thread specific variable with static storage duration. Consider the following C code:

int global\_x;

foo() {

int local\_x

static int static\_x = 0;

…

}

global\_x and static\_x are allocated once per process and all threads share the same instance. Whereas local\_x is allocated in stack and each thread gets its own stack and hence the auto variable local\_x is thread specific. There is no easy way to define a per thread global/static variable. POSIX thread interface allows creating thread specific static storage variable using pthread getspecific and pthread setspecific. But this interface is cumbersome to use.

To solve this, Thread Local Storage (TLS) is defined as a class of storage that allows defining thread specific variable with static storage durations. A TLS variable or thread-local variable is a global/static variable that is instanced once per thread.

This document describes the TLS specification for the C6000 family of processors and will become part of the C6000 EABI specification. TLS is only supported under C6000 EABI and not supported under C6000 COFF ABI.

The way a TLS variable is accessed depends on how the OS or run-time creates and manages thread local storage for each thread. Linux systems need to support TLS allocation for multiple dynamic libraries and libraries loaded during runtime using dlopen(). Also, Linux systems may require allocating TLS storage lazily only when the thread-local is accessed. This requires a sophisticated TLS storage management and affects how the thread-local is accessed. On the other hand, a static executable that includes an RTOS needs only to manage a single TLS block and the access can be simple.

This document first describes how thread-locals are specified in source code and how they are represented in the ELF object file. Then it describes the run-time representation of thread-locals for C6x Linux, static executable and bare-metal dynamic linking TLS models and how the thread-locals are accessed.

# Terms and Concepts

Thread-local variables are thread specific and have static storage duration. They must be allocated similar to global/static variables that are allocated to .data if initialized and .bss uninitialized. Global/static variables need only one copy per process whereas thread-locals need an instance per each thread. This means, as part of thread creation, the storage for thread-locals must be allocated and initialized. This also means there should be an initialization image that can be used for initializing per thread TLS storage. The output of the static linker, the static link unit, must contain a TLS initialization image if thread local is used. The static link unit is referred to as **module** in this document. TLS initialization image from a single module is referred as **TLS Image** in this document. TLS is allocated for each thread as part of the thread creation and initialized with the data from TLS Image. The memory allocated per thread for thread-locals from a single module is referred as **TLS** **Block** in this document.

In the static executable model the static linker produces an exe which is loaded and executed from the start address. RTOS and/or threads library are linked-in as part of the executable. In this case there is only one module and hence only one TLS Image and TLS Block. This makes TLS access very simple. The main thread is usually created during boot up and the other threads are created by the threads library. The main thread’s TLS Block should be allocated and initialized by boot routine. It is the responsibility of the thread library to allocate and initialize TLS for the threads it creates.

In the C6x Linux system a program (process) is created by loading multiple modules, an executable and zero or more dynamic libraries. Each module can have a TLS Image. The program’s TLS Image consists of all the modules’ TLS Images. This is called **TLS Template**. Normally, the executable and all dependent modules are loaded at process startup. These are called **initially loaded modules.** A Linux program can also load a dynamic library after startup by calling dlopen() system function. The modules loaded after startup is referred as **dlopened modules**. During thread creation, TLS blocks are created based on the TLS Template. This run-time structure, consisting of TLS Blocks from all the modules is referred as **TLS**.

In the case of bare-metal dynamic linking, by default, there are only initially loaded modules and they can be consecutively placed to form the TLS Template.

Refer to [1] for more information on C6x Program loading and dynamic linking.

# User Interface

Over the years, programming languages have been extended to allow definition of thread-local variables:

* Compilers for Linux systems (GCC, Sun, IBM, and Intel) support **\_\_thread** storage qualifier as a C/C++ language extension. This is not an official language extension, however.
* Compilers for Windows (MS VC++, Intel, Borland) support **\_\_declspec(thread)**.
* The latest C++ standard, C++11 (ISO/IEC 14882:2011), introduces **thread\_local** storage class specifier.
* C1X, the next C standard currently being defined, is considering **\_Thread\_local** storage class specifier.

The language extension used to support thread-local is Q-o-I.

Thread-local variables can be initialized or uninitialized. Uninitialized thread-locals are initialized to zero similar to uninitialized global/static variables.

# ELF Object File Representation

Refer to the ELF specification [2] for details on how thread local storage is represented in ELF relocatable object files and ELF modules.

To summarize, thread-local variables are represented in the object files and ELF modules similar to the static data. The difference is that ELF requires that thread local variables are allocated in sections with **SHF\_TLS** flag set in relocatable files. Also, ELF suggests the section name “.tdata” and “.tbss” for initialized and uninitialized thread-locals respectively. These sections have read-write permission. In modules, ELF requires that TLS segment is indicated by **PT\_TLS** segment type. This segment is read-only. The PT\_TLS segment is the TLS Image.

Thread-local symbols have symbol type **STT\_TLS**.

# TLS Access Models

Each thread has an instance of the thread-local variable. An access to a thread-local should access the current thread’s instance of the thread local. This means, the thread-local access needs to find the current thread’s TLS and access the variable using an offset into the TLS block where the variable is defined.

The C6x Linux TLS access model needs to satisfy more constraints and can be complex and needs to conform to already established conventions. The static exe access, on the other hand is simple as there is only one TLS block and any thread-local can be accessed using TP relative addressing. It is useful to first describe the more complex C6x Linux TLS model and then describe the static exe TLS model.

## C6x Linux TLS models

In some dynamic linking models, including Linux, a module can be loaded during run-time using dlopen. The TLS block from the dlopened module cannot be allocated at a fixed offset from TP for all the threads. Hence the access to thread local is by reference using the module identifier and the offset of the thread-local in the module’s TLS block.

Figure 5.1 shows the C6x Linux TLS run-time representation. Each thread has an instance of this run-time TLS structure.

Figure 5.1 C6x Linux TLS Run-Time Representation

For each thread, the thread pointer TP points to the Thread Control Block (TCB). The executable’s TLS block, if exists, is placed after the TCB after adjusting for the alignment. TLS blocks from other initial modules are placed subsequently honoring their alignment requirement. The TCB and the following initial modules’ TLS blocks constitute the program’s static TLS. The static TLS for a thread is created as part of the thread creation.

The TCB is 64-bits wide. The first 32-bits points to the Dynamic Thread Vector (dtv). The remaining 32-bits are reserved.

The dtv is a vector of 32-bit size elements. dtv[0] is generation id. This is used to manage the dynamic growth of the dtv as more dlopened modules are loaded. dtv[n] where n != 0 is a 32-bit pointer to the TLS block of module n. When a module with TLS data is loaded a module id is assigned for that module. This module id is process specific. A dynamic shared library shared by multiple processes can have different module ids in each process. Module id 1 is always assigned to the executable.

**Note**: dtv is needed only to enable accessing thread locals from dlopened modules. If a system prohibits dlopen then dtv can be completely eliminated. If dtv is eliminated then TCB can also be eliminated.

The main thread is created by the dynamic loader and the subsequent threads are created by the thread library. When the main thread is created, the dtv only need to contain the pointers to the initially loaded modules.

When a thread dlopens a new module, the modules TLS block should be allocated for all the threads in the process. This is needed in case the other threads access this new module’s thread-local data. However, allocating the TLS block of the dlopened module can be deferred until the first time it is accessed. This can be done by initializing dtv[dlopened-module-id] to TLS\_DTV\_UNALLOCATED. \_\_tls\_get\_addr() can check if dtv[module\_id] is TLS\_DTV\_UNALLOCATED and if so allocates and initializes the TLS block for the current thread.

There are four TLS access models discussed in the literature [3, 4] under Linux dynamic linking model. These models are widely used. These are:

### General Dynamic TLS Access Model

This is the most generic TLS model. Objects using this access model can be used to build any Linux module: executable, initially loaded modules and dlopened modules. The generated code for this model cannot assume the module-id or the offset is known during static linking. The compiler generates a call to \_\_tls\_get\_addr() to get the address of the thread-local. The module-id and the thread-local’s offset in the module’s TLS block are passed as parameters. The code obtains the module-id and offset from GOT entries to ensure PIC and symbol preemption.

#### General Dynamic TLS Addressing

\_\_tls\_get\_addr() needs to receive module-id and offset as arguments. These are loaded from GOT. The simple way is pass the module-id and offset as two arguments:

void \* \_\_tls\_get\_addr(unsigned int module\_id, ptrdiff\_t offset);

Note that both the arguments are 32-bits and the GOT entries are also 32-bit entries. As an optimization we can load these two GOT entries as a 64-bit double word if the ISA supports. To do this, the two GOT entries must be allocated consecutively and aligned to 64-bit boundary. This GOT entity can be thought of as a struct:

struct TLS\_descriptor

{

unsigned int module\_id;

ptrditt\_t offset;

} \_\_attribute\_\_ ((aligned (8)));

Then the \_\_tls\_get\_addr() interface becomes:

void \* \_\_tls\_get\_addr(struct TLS\_descriptor);

In EABI a struct of size 64-bits or less is passed by value. The above interface results in passing the TLS descriptor in A5:A4 register pair. In little-endian mode module-id is passed in A4 and the offset is in A5. In big-endian mode the registers are swapped as per the C6x EABI calling conventions. The examples in this spec uses little-endian.

Using this interface, the thread-local access becomes (C64 and above):

LDDW \*+DP($GOT\_TLS(X)), A5:A4 ;reloc R\_C6000\_SBR\_GOT\_U15\_D\_TLS

|| CALLP \_\_tls\_get\_addr,B3 ; A4 has the address of X at return

LDW \*A4, A4 ; A4 has the value of X

The relocation R\_C6000\_SBR\_GOT\_U15\_D\_TLS causes the linker to create GOT entries for module-id and offset for x as shown below:

64-bit aligned address:

GOT[n] ;reloc R\_C6000\_TLSMOD (symbol X)

GOT[n+1] ;reloc R\_C6000\_TBR\_U32 (symbol X)

The linker then resolves the R\_C6000\_SBR\_GOT\_U15\_D\_TLS with the DP relative offset of the above GOT entity. The dynamic loader resolves R\_C6000\_TLSMOD to the module-id of the module where x is defined. It resolves R\_C6000\_TBR\_U32 to the offset of x in the module’s TLS block.

This interface requires ‘LDDW B14/B15, 15-bit offset, regpair’ instruction. At present, C6x ISAs don’t support such instructions. Yet, there could be support for such instructions in the future so we define the \_\_tls\_get\_addr() interface as

void \* \_\_tls\_get\_addr(struct TLS\_descriptor);

This specification requires that the linker allocate the GOT entries of a thread-local’s module-id and offset consecutively and align the first entry to 64-bits when R\_C6000\_SBR\_GOT\_U15\_D\_TLS relocation is found.

For now, since the present C6x ISAs don’t support 64-bit load store using 15-bit DP offset, the following code sequence can be used (little-endian):

LDW \*+DP($GOT\_TLSMOD(X)), A5 ;reloc R\_C6000\_SBR\_GOT\_U15\_W\_TLSMOD

LDW \*+DP($GOT\_TBR(X)), A4 ;reloc R\_C6000\_SBR\_GOT\_U15\_W\_TBR

|| CALLP \_\_tls\_get\_addr,B3 ; A4 has the address of X at return

LDW \*A4, A4 ; A4 has the value of X

The relocations R\_C6000\_SBR\_GOT\_U15\_W\_TLSMOD and R\_C6000\_SBR\_GOT\_U15\_W\_TBR cause the linker to create GOT entries for module-id and offset respectively for x. This access mode doesn’t require these GOT entries to be consecutive and 64-bit aligned. If the linker doesn’t also see DW\_TLS relocation for the same symbol, it is free to define the module-id and offset GOT entries separately without 64-bit alignment. However, if it sees DW\_TLS in addition to the TLSMOD/TBR relocations for the same symbol, 64-bit aligned consecutive GOT entries must be defined and reused for the TLSMOD/TBR relocations.

[ Note: Since the C6x ISA don’t support 64-bit load using 15-bit DP offset, the relocation R\_C6000\_SBR\_GOT\_U15\_D\_TLS is not useful now. Hence this relocation is not defined.]

If the GOT must be addressed using far-DP addressing, then the general dynamic addressing becomes:

MVKL $DPR\_GOT\_TLSMOD(X), A5 ;reloc R\_C6000\_SBR\_GOT\_L16\_W\_TLSMOD

MVKH $DPR\_GOT\_TLSMOD(X), A5 ;reloc R\_C6000\_SBR\_GOT\_H16\_W\_TLSMOD

ADD DP, A5, A5

LDW \*A5, A5

MVKL $DPR\_GOT\_TPR(X), A4 ;reloc R\_C6000\_SBR\_GOT\_L16\_W\_TBR

MVKH $DPR\_GOT\_TPR(X), A4 ;reloc R\_C6000\_SBR\_GOT\_H16\_W\_TBR

ADD DP, A4, A4

LDW \*A4, A4

|| CALLP \_\_tls\_get\_addr,B3 ; A4 has the address of X at return

LDW \*A4, A4 ; A4 has the value of X

\_\_tls\_get\_addr() can calculate the thread-local address as shown below:

void \* \_\_tls\_get\_addr(struct TLS\_descriptor desc)

{

void \*TP = \_\_c6xabi\_get\_tp();

int \*dtv = (int\*)(((int\*) TP)[0]);

char \*tls = (char \*)dtv[desc.module\_id];

return tls + desc.offset;

}

### Local Dynamic TLS Access Model

This is an optimization of General Dynamic Model to access own data. If the compiler knows it is accessing module’s own thread-local then this model can be used. If the thread-local is defined in the same module where it is accessed then the TLS offset is known at static link time. Still the module id is not known during the static link time. A call to \_\_tls\_get\_addr() with offset parameter zero returns the base address of the module’s TLS block. This base address can be used to access all the own thread-local data in the module.

During compile time, own data is identified using the symbol binding and visibility. Symbols with static scope or hidden/protected visibility are own data. In this model own thread-local x can be accessed as shown below:

LDW \*+DP($GOT\_TLSMOD(x)), A4 ; reloc R\_C6000\_SBR\_GOT\_U15\_W\_TLSMOD

MVK $TBR\_word(x), A5 ; reloc R\_C6000\_TBR\_U15\_W

|| CALLP \_\_tls\_get\_addr,B3 ; A4 has the address of x at return

As mentioned above, the own TLS base can be obtained once and reused to access other own thread-locals as shown below:

LDW \*+DP($GOT\_TLSMOD()), A4 ; reloc R\_C6000\_SBR\_GOT\_U15\_W\_TLSMOD w/ Symbol=0

MVK 0x0, A5 ;

|| CALLP \_\_tls\_get\_addr,B3 ; A4 has the module’s own TLS base

MVK $TBR\_byte(x), A5 ; reloc R\_C6000\_TBR\_U15\_B; Get x’s scaled TLS offset

LDB \*A4[A5], A6 ; A6 has the value of thread-local char x

MVK $TBR\_hword(y), A5 ; reloc R\_C6000\_TBR\_U15\_H; Get y’s scaled TLS offset

LDH \*A4[A5], A6 ; A6 has the value of thread-local short y

MVK $TBR\_word(z), A5 ; reloc R\_C6000\_TBR\_U15\_W; Get z’s scaled TLS offset

LDW \*A4[A5], A6 ; A6 has the value of thread-local int z

MVK $TBR\_dword(l), A5 ;reloc R\_C6000\_TBR\_U15\_D; Get l’s scaled TLS offset

LDDW \*A4[A5], A7:A6 ;A7:A6 has the value of thread-local long long l

The relocation R\_C6000\_SBR\_GOT\_U15\_W\_TLSMOD resolves to own module’s module-id when the symbol is zero. The TBR\_U15 relocations encode 15-bit unsigned offset from the module’s TLS Base for near TB (TLS Block Base) addressing. They are scaled according to the access width. The above addressing can access TLS block of size 32K. This specification limits the size of each module’s TLS block to 32K as this limit is expected to be sufficient for most use cases. Hence the far TB relative address is not defined. It is easy to define the far TBR addressing but it will use up 8 new relocations and it is better to conserve the limited number of relocations (256) ELF allows.

The static linker resolves all the TBR relocations and they are static-only relocation. That is, these relocations cannot be in the dynamic relocation table.

### Initial Exec TLS Access Model

Objects used to build initially loaded modules can use this access model. Modules using this access model cannot be dlopened. Since the module will always be initially loaded and the dynamic loader can allocate TLS blocks from initial modules consecutively after the executable’s TLS block, the offset from the thread pointer is known at dynamic link time. The thread local can be accessed using \*(TP + offset) where the offset is loaded from GOT to ensure PIC and symbol preemption. Modules built with this addressing cannot be dlopned. Such modules are marked with the dynamic flag DF\_STATIC\_TLS. The dynamic loader will refuse to dlopen modules marked DF\_STATIC\_TLS.

#### Thread Pointer

This addressing needs a way to obtain the thread pointer of the current thread. A new c6xabi function, \_\_c6xabi\_read\_tp(), is defined which returns the thread pointer value for the current thread. This function doesn’t modify any register other than the return register A4. This function can be called via PLT and hence the caller should assume B30 and B31 are modified by the call to this function. This function has the following signature:

void \* \_\_c6xabi\_get\_tp(void);

The thread library is responsible for providing a definition of this function.

#### Initial Exec TLS Addressing

In the Initial Exec model, the thread-local is accessed as shown below:

callp \_\_c6xabi\_get\_tp() ;Returns TP in A4; Can be CSEed

LDW \*+DP($GOT\_TPR\_byte(x)), A5 ;reloc R\_C6000\_SBR\_GOT\_U15\_W\_TPR\_B

LDB \*A4[A5], B4 ;

LDW \*+DP($GOT\_TPR\_hword(x)), A5 ;reloc R\_C6000\_SBR\_GOT\_U15\_W\_TPR\_H

LDH \*A4[A5], B4 ;

LDW \*+DP($GOT\_TPR\_word(x)), A5 ;reloc R\_C6000\_SBR\_GOT\_U15\_W\_TPR\_W

LDW \*A4[A5], B4 ;

LDW \*+DP($GOT\_TPR\_dword(x)), A5 ;reloc R\_C6000\_SBR\_GOT\_U15\_W\_TPR\_D

LDDW \*A4[A5], B4 ;

The relocation R\_C6000\_SBR\_GOT\_U15\_W\_TPR\_[B|H|W] causes the linker to create a GOT entry for x’s TP relative offset:

GOT[m] ;reloc R\_C6000\_TPR\_U32\_B (symbol x)

GOT[n] ;reloc R\_C6000\_TPR\_U32\_H (symbol y)

GOT[o] ;reloc R\_C6000\_TPR\_U32\_W (symbol z)

GOT[p] ;reloc R\_C6000\_TPR\_U32\_D (symbol z)

The \_TPR\_U32\_[B|H|W|DW] relocations are resolved by the dynamic loader with the offset of x from the thread pointer. These relocations are scaled as per the access width.

If GOT must be accessed using far-DP addressing the sequence is as shown below:

callp \_\_c6xabi\_get\_tp() ;Returns TP in A4; Can be CSEed

MVKL $DPR\_GOT\_TPR\_byte(x), A5 ;reloc R\_C6000\_SBR\_GOT\_L16\_W\_TPR\_B

MVKH $DPR\_GOT\_TPR\_byte(x), A5 ;reloc R\_C6000\_SBR\_GOT\_H16\_W\_TPR\_B

ADD DP, A5, A5

LDW \*A5, A5

LDB \*A4[A5], A6

MVKL $DPR\_GOT\_TPR\_hword(x), A5 ;reloc R\_C6000\_SBR\_GOT\_L16\_W\_TPR\_H

MVKH $DPR\_GOT\_TPR\_hword(x), A5 ;reloc R\_C6000\_SBR\_GOT\_H16\_W\_TPR\_H

ADD DP, A5, A5

LDW \*A5, A5

LDH \*A4[A5], A6

MVKL $DPR\_GOT\_TPR\_word(x), A5 ;reloc R\_C6000\_SBR\_GOT\_L16\_W\_TPR\_W

MVKH $DPR\_GOT\_TPR\_word(x), A5 ;reloc R\_C6000\_SBR\_GOT\_H16\_W\_TPR\_W

ADD DP, A5, A5

LDW \*A5, A5

LDW \*A4[A5], A6

MVKL $DPR\_GOT\_TPR\_dword(x), A5 ;reloc R\_C6000\_SBR\_GOT\_L16\_W\_TPR\_D

MVKH $DPR\_GOT\_TPR\_dword(x), A5 ;reloc R\_C6000\_SBR\_GOT\_H16\_W\_TPR\_D

ADD DP, A5, A5

LDW \*A5, A5

LDDW \*A4[A5], A6

### Local Exec TLS Model

This is an optimization of Initial Exec Model. When the program’s initial TLS image (normally called static TLS image) is created, the TLS block is always placed at a known offset from thread pointer. Normally this is the TCB plus the TLS Block Base offset. Hence the executable’s own thread-local has a thread pointer relative offset which is a static link time constant. Thread-locals in this case can be accessed using inline constant offset and a GOT entry is not needed. Objects using this access model cannot be used to build dynamic library.

CALLP \_\_c6xabi\_get\_tp() ; Returns TP in A4. Can be CSEed.

MVK $TPR\_byte(x), A5 ; reloc R\_C6000\_TPR\_U15\_B

LDB \*A4[A5], A4 ; A4 contains the value of thread-local char x

MVK $TPR\_hword(y), A5 ; reloc R\_C6000\_TPR\_U15\_H

LDH \*A4[A5], A4 ; A4 contains the value of thread-local short y

MVK $TPR\_word(z), A5 ; reloc R\_C6000\_TPR\_U15\_W

LDW \*A4[A5], A4 ; A4 contains the value of thread-local int z

MVK $TPR\_dword(l), A5 ; reloc R\_C6000\_TPR\_U15\_D

LDDW \*A4[A5], A7:A6 ; A7:A6 contains the value of thread-local long long l

The TPR\_U15 relocations encode 15-bit unsigned TP relative offsets (offset from the address TP point to) for near TP relative addressing. They are scaled according to the access width. The above addressing can access TLS block of size 32K. This specification limits the size of the total static TLS to 32K as this limit is expected to be sufficient for most use cases. Hence the far TP relative address is not defined. It is easy to define the far TPR addressing but it will use up 8 new relocations and it is better to conserve the limited number of relocations (256) ELF allows.

## Static Executable TLS model

The static executable TLS model can be supported by a C6x EABI conforming compiler as a Q-o-I. It is not required for C6x EABI compliance.

In the case of static executable, there is only one TLS block and the TLS offset of each thread-local is known during static link time. The access to thread-local is \*(TLS base + offset). Figure 5.2 shows the run-time layout of the TLS. TP is the thread pointer which points to the current thread’s TLS block. Offset of x is known during static linking.

Figure 5.2. Static Executable TLS Run-Time Representation

### Static Executable Addressing

The thread-local access code in this case is same as Linux Local Exec model. In the case of the static exe there is no TCB and hence the TP relative offset is same as TLS Block Base relative address. Ideally we can generate TBR addressing for this case. However, the compile options can be used to build bare-metal dynamic linking model and it needs TCB. So, we generate the TP relative addressing for the static exe model as shown below:

CALLP \_\_c6xabi\_get\_tp() ; Returns TP in A4. Can be CSEed.

MVK $TPR\_byte(x), B4 ; reloc R\_C6000\_TPR\_U15\_B

LDB \*A4[B4], A4 ; A4 contains the value of thread-local char x

MVK $TPR\_hword(y), B4 ; reloc R\_C6000\_TPR\_U15\_H

LDH \*A4[B4], A4 ; A4 contains the value of thread-local short y

MVK $TPR\_word(z), B4 ; reloc R\_C6000\_TPR\_U15\_W

LDW \*A4[B4], A4 ; A4 contains the value of thread-local int z

MVK $TPR\_dword(l), B4 ; reloc R\_C6000\_TPR\_U15\_D

LDDW \*A4[B4], A7:A6 ; A4 contains the value of thread-local int l

The TPR relocations are resolved by the static linker with the offset of the variable in the executable’s TLS block. The far TPR relocations can be used if the TLS block is expected to be bigger than 32K.

### Static exe TLS Runtime Architecture

In dynamic linking systems the dynamic loader creates the main thread and the thread library creates additional threads. As part of the main thread creation, the dynamic loader allocates and initializes the main thread’s TLS. Also, the dynamic loader can easily find the TLS initialization image using the segment type.

In the case of static executable, there is no dynamic loader to take up the above roles. The static linking model should support the following requirements:

* The allocation and initialization of the main thread’s TLS before main() or any user code from init\_array is called.
* During the main thread’s execution, \_\_c6xabi\_get\_tp() should return the pointer to main thread’s TLS. This function must be supported even when there is no thread library.
* The thread library should have a way to access the TLS initialization image so that it can initialize the TLS blocks for the threads it creates.

### Static Exe TLS allocation

There are three memory areas need to be allocated to support TLS: The initialization image, the main thread’s TLS block and the TLS area where the thread-library can allocate TLS blocks for the threads it creates.

#### TLS Initialization Image Allocation

The TLS initialization image is created in the output section .TI.tls\_init. This section is read-only. The user can specify the allocation for this output section as below:

.TI.tls\_init > ROM

If no allocation is specified, this output section is allocated using .cinit allocation. If no allocation specified for .cinit the default allocation is used. The user cannot specify the section specifier for this section.

The .TI.tls\_init output segment is formed by combining the following linker created components:

1. **.tdata.load** - Compressed TLS initialized section
2. **.tbss.load** - Zero-init section to zero initialize uninitialized section
3. **\_\_TI\_tls\_init\_table** - Copy table to initialize TLS blocks. This copy table has two copy records. One for each of the above initialization sections.

#### Main Thread’s TLS Allocation

The user can specify the allocation for the main threads TLS block using:

.TI.tls > RAM

This uninitialized output section is initialized using the \_\_TI\_tls\_init\_table copy table during boot time. The user cannot specify the section specifier for this section.

If no allocation is specified for this section, it is allocated using .fardata output section’s allocation. If no allocation specified for .fardata, .far allocation is used. Otherwise, default allocation is used.

The linker defines the symbol \_\_TI\_TLS\_MAIN\_THREAD\_BASE to point to the start of the .TI.tls output section.

#### Thread Library’s TLS Region Allocation

Allocating the TLS region to be used by the thread-library is specific to the library. The specification doesn’t dictate a specific way for this. One possible way is to do the following:

.tls\_region { . += 0x2000; } START(TLS\_REGION\_START) > RAM

The thread library can use the symbol TLS\_REGION\_START to locate the TLS region. User might want to allocation TLS blocks for N number of threads and it is useful to know the size of TLS block. The user can do the following:

.tls\_region { . += MAX\_THREADS \* \_\_TI\_TLS\_BLOCK\_SIZE; } > RAM

The static linker defines the symbol \_\_TI\_TLS\_BLOCK\_SIZE and sets it to the size of the TLS block.

### Main Thread’s TLS Initialization

During boot, the startup code calls the RTS function \_\_TI\_tls\_init(NULL) to initialize the main thread’s TLS block. The RTS function initializes the main thread’s TLS if NULL argument is passed.

### TLS Initialization by Thread Library

The thread library must initialize the TLS blocks once it creates them for a given thread. The static exe TLS model defines a new RTS function for this.

\_\_TI\_tls\_init(void \* dest\_addr);

The thread library must pass the address of the TLS block to be initialized to this function.

This RTS function uses the copy table to perform the initialization. However, how this function initializes the TLS block is based on the interface between the static linker and this RTS function which can change in the future. So, the thread library must only rely on this RTS function interface to initialize the TLS blocks.

### Thread Pointer

In the Static Executable TLS model the function \_\_c6xabi\_get\_tp() is called to get the thread pointer value of the current thread. If a thread library is used, it is responsible for providing this function. The thread library knows the address of the TLS block for the threads it creates. However, the main thread is not created by it so it needs a standard way to find the address of the main thread’s TLS block. As mentioned above, the static linker defines the symbol \_\_TI\_TLS MAIN\_THREAD\_BASE for this purpose.

The TI RTS provides the following definition for the \_\_c6xabi\_get\_tp() function:

extern \_\_attribute\_\_((weak)) far const void \* \_\_TI\_TLS\_MAIN\_THREAD\_Base;

\_\_attribute\_\_((weak)) void \* \_\_c6xabi\_get\_tp(void)

{

return &\_\_TI\_TLS\_MAIN\_THREAD\_Base;

}

This function is defined ‘weak’ so that a strong definition from the thread library will be used if present.

Let us consider the unlikely case where a user uses thread-locals but doesn’t include thread library. Obviously they cannot create any new threads. But the main thread should work and the main thread’s thread-local should be accessible. In such cases, the above mentioned RTS function is linked in and provides access to the main thread’s TLS.

## Bare-Metal Dynamic Linking TLS Model

In general the bare-metal dynamic linking only involves modules loaded initially. dlopen is not presently supported in bare-metal dynamic linking. However, we cannot rule this out in the future. So, the TLS access model should accommodate future support for dlopen. As we have seen in the Linux TLS model, the TLS access for dlopened module is through dtv. The run-time representation for bare-metal is same as the Linux run-time model.

At present the objects compiled for static executable can be used to create bare-metal dynamic exe/lib. It is beneficial to maintain this use case even when TLS is used. A compile time flag would be required to compile code to be used in a dlopened module. That is, by default, the code generated is for the bare-metal model that doesn’t support dlopen.

### Default TLS Addressing for Bare-Metal Dynamic Linking

The default code generation for TLS should work for both static exe and bare-metal dynamic linking. For static exe, we generate the following addressing using TPR addressing:

CALLP \_\_c6xabi\_get\_tp() ; Returns TP in A4. Can be CSEed.

MVK $TPR\_byte(x), B4 ; reloc R\_C6000\_TPR\_U15\_B

LDB \*A4[B4], A4 ; A4 contains the value of thread-local char x

The code generated by default for bare-metal dynamic linking can assume that all modules are initially loaded. This means the offset of thread-locals is a dynamic link time constant as shown in Figure 5.3. Hence the TPR addressing can be used. The only difference is that in the bare-metal dynamic linking case, a 64-bit TCB is needed to make the code compatible with any future support for dlopen. In the case of static exe the TCB is not present. Still the TPR addressing can be used for both the models. The static linker will use TCB size of zero for static exe and TCB size of 64-bits for bare-metal dynamic linking.

As mentioned earlier, the initially loaded modules are placed consecutively and the executable’s TLS block comes after the TCB. In this case, the variables in the executable can be accessed using static link time constant offsets from TP. The variables defined in the dynamic libraries can be accessed using dynamic link time constant offsets from TP.

When this addressing is generated the modules are marked DF\_STATIC\_TLS.

When building a dynamic executable, the static linker resolves the TPR relocations for symbols defined in the executable (own data) to the TP offset. If the symbol is imported, the relocation is copied to the dynamic relocation table to be resolved by the dynamic loader. When building dynamic library, the TPR relocations are copied to the dynamic relocation table.

The thread-local access in bare-metal can result in dynamic relocations in code segment. This means the resulting module is not true-PIC. TI compiler supports bare-metal PIC under --gen\_pic option. When this option is used, the TPR offsets should be accessed from GOT entry to generate position independent code.

Figure 5.3. Bare-Metal Default TLS Run-Time Representation

### TLS block creation

In the case of bare-metal dynamic linking system, the dynamic loader is responsible for creating the main thread’s TLS block. The dynamic loader when loading an ELF File should load the PT\_TLS segment and should provide a way for the thread library to access the PT\_TLS Initialization Image so the thread library can use it to initialize the TLS blocks for the threads it creates. The static linker when building the dynamic executable/library generates the PT\_TLS segment as per ELF requirements.

# Thread-Local Symbol Resolution

A Thread-local reference can only be resolved by a thread local definition. The linker should enforce this requirement. Also, the presence of thread-local definition and a normal global definition with the same name is an error.

Thread-locals can be defined or declared weak. A weak thread-local definition implies that it can be overridden by a strong definition if available. If a strong definition is not found, the weak definition is used. No special care is needed to support thread-local weak definitions.

A weak thread-local symbol reference is resolved to zero address if a definition is not found. This requires special handling in each TLS Addressing Models.

## General/Local Dynamic TLS Weak Reference Addressing

In both the General and Local Dynamic TLS model, the function \_\_tls\_get\_addr() is called to get the thread-local’s address. The module-id in both General and Local Dynamic TLS models are obtained from GOT. The offset is obtained from GOT in General Dynamic model and as a static link-time constant in Local Dynamic model. In the case of weak undefined reference, there is no thread-local definition to resolve the weak reference. Since there is no definition, the module-id and TBR offset resolves to zero.

For weak thread-local references there is no change in the code generated to access them. The R\_C6000\_TLSMOD relocation and all the R\_C6000\_TBR relocations resolve to zero if there if the thread-local reference is weak and there is no definition.

The \_\_tls\_get\_addr() function returns zero when the module-id and offset are zero. This ensures that the undefined weak reference address is resolved to zero.

## Initial and Local Exec TLS Weak Reference Addressing

In these models TPR addressing is generated to access thread-local. There is no way to ensure the undefined weak reference resolves to address zero using this accessing mode. Hence, if the symbol reference is weak, in the Initial Exec models the General Dynamic Model addressing is generated and in Local Exec model the Local Dynamic Model addressing is generated. This way, the \_\_tls\_get\_addr() will return zero for undefined weak references.

## Static TLS Model Weak References

In the static executable and bare-metal dynamic linking model the TPR addressing is generated. Since the TPR addressing cannot ensure the address of zero for weak undefined references and cannot be used. An addressing similar to General/Local Dynamic model is required to support weak references.

In static and bare-metal dynamic linking the following addressing is generated for weak references.

MVK $TPR\_S16(x), A5 ; reloc R\_C6000\_TPR\_S16

|| CALLP \_\_c6xabi\_get\_addr,B3 ; A4 has the address of x at return

The C6x eabi function \_\_c6xabi\_get\_addr() has the following signature:

void \* \_\_c6xabi\_get\_addr(ptrdiff\_t TPR\_offst);

This function accepts 32-bit TPR offset and returns the address of the thread-local. A special value of -1 is used to indicate a weak undefined reference and a zero is returned in this case.

The static linker and dynamic linker resolve TPR\_S16 relocations to -1 for a weak undefined reference.

# Thread-Local Relocations

The table below lists the new relocations added to support thread-locals. The following notations are used in the table:

|  |  |
| --- | --- |
| TBR(x) | The offset of x from the TLS Block Base. |
| TPR(x) | The offset of x from the thread-pointer TP. |
| TLS(x) | TLS Descriptor for x which contains the module-id and TBR offset of x. |
| TLSMOD(x) | The module identifier of the module that defines x. |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Value** | **Operation** | **Constraints** |
| R\_C6000\_TBR\_U15\_B | 33 | TBR(S) | Static only |
| R\_C6000\_TBR\_U15\_H | 34 | TBR(S) | Static only |
| R\_C6000\_TBR\_U15\_W | 35 | TBR(S) | Static only |
| R\_C6000\_TBR\_U15\_D | 36 | TBR(S) | Static only |
| R\_C6000\_TPR\_S16 | 37 | TBR(S) |  |
| R\_C6000\_TPR\_U15\_B | 38 | TPR(S) |  |
| R\_C6000\_TPR\_U15\_H | 39 | TPR(S) |  |
| R\_C6000\_TPR\_U15\_W | 40 | TPR(S) |  |
| R\_C6000\_TPR\_U15\_D | 41 | TPR(S) |  |
| R\_C6000\_TPR\_U32\_B | 42 | TPR(S) | Dynamic only |
| R\_C6000\_TPR\_U32\_H | 43 | TPR(S) | Dynamic only |
| R\_C6000\_TPR\_U32\_W | 44 | TPR(S) | Dynamic only |
| R\_C6000\_TPR\_U32\_D | 45 | TPR(S) | Dynamic only |
| R\_C6000\_SBR\_GOT\_U15\_W\_TLSMOD | 46 | GOT(TLSMOD(S)) + A - B | Static only |
| R\_C6000\_SBR\_GOT\_U15\_W\_TBR | 47 | GOT(TBR(S)) + A - B | Static only |
| R\_C6000\_SBR\_GOT\_U15\_W\_TPR\_B | 48 | GOT(TPR(S))+A-B | Static only |
| R\_C6000\_SBR\_GOT\_U15\_W\_TPR\_H | 49 | GOT(TPR(S))+A-B | Static only |
| R\_C6000\_SBR\_GOT\_U15\_W\_TPR\_W | 50 | GOT(TPR(S))+A-B | Static only |
| R\_C6000\_SBR\_GOT\_U15\_W\_TPR\_D | 51 | GOT(TPR(S))+A-B | Static only |
| R\_C6000\_SBR\_GOT\_L16\_W\_TLSMOD | 52 | GOT(TLSMOD(S)) + A - B | Static only |
| R\_C6000\_SBR\_GOT\_L16\_W\_TBR | 53 | GOT(TBR(S)) + A - B | Static only |
| R\_C6000\_SBR\_GOT\_L16\_W\_TPR\_B | 54 | GOT(TPR(S))+A-B | Static only |
| R\_C6000\_SBR\_GOT\_L16\_W\_TPR\_H | 55 | GOT(TPR(S))+A-B | Static only |
| R\_C6000\_SBR\_GOT\_L16\_W\_TPR\_W | 56 | GOT(TPR(S))+A-B | Static only |
| R\_C6000\_SBR\_GOT\_L16\_W\_TPR\_D | 57 | GOT(TPR(S))+A-B | Static only |
| R\_C6000\_SBR\_GOT\_H16\_W\_TLSMOD | 58 | GOT(TLSMOD(S)) + A - B | Static only |
| R\_C6000\_SBR\_GOT\_H16\_W\_TBR | 59 | GOT(TBR(S)) + A - B | Static only |
| R\_C6000\_SBR\_GOT\_H16\_W\_TPR\_B | 60 | GOT(TPR(S))+A-B | Static only |
| R\_C6000\_SBR\_GOT\_H16\_W\_TPR\_H | 61 | GOT(TPR(S))+A-B | Static only |
| R\_C6000\_SBR\_GOT\_H16\_W\_TPR\_W | 62 | GOT(TPR(S))+A-B | Static only |
| R\_C6000\_SBR\_GOT\_H16\_W\_TPR\_D | 63 | GOT(TPR(S))+A-B | Static only |
| R\_C6000\_TLSMOD | 64 | TLSMOD(S) | Dynamic only |
| R\_C6000\_TBR\_U32 | 65 | TBR(S) | Dynamic only |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Signedness** | **Field  [CS, O, FS]** | **Addend (A)** | **Result (R)** | **Overflow Check** | **Encoded Value (EV)** |
| R\_C6000\_TBR\_U15\_B | unsigned | [32,8,15] | ZE(F) | TBR(S) | Yes | R |
| R\_C6000\_TBR\_U15\_H | unsigned | [32,8,15] | ZE(F<<1) | TBR(S) | Yes | R >> 1 |
| R\_C6000\_TBR\_U15\_W | unsigned | [32,8,15] | ZE(F<<2) | TBR(S) | Yes | R >> 2 |
| R\_C6000\_TBR\_U15\_D | unsigned | [32,8,15] | ZE(F<<3) | TBR(S) | Yes | R >> 3 |
| R\_C6000\_TPR\_S16 | Signed | [32,7,16] | SE(F) | TBR(S) | Yes | R |
| R\_C6000\_TPR\_U15\_B | Unsigned | [32,8,15] | ZE(F) | TPR(S) | Yes | R |
| R\_C6000\_TPR\_U15\_H | Unsigned | [32,8,15] | ZE(F<<1) | TPR(S) | Yes | R >> 1 |
| R\_C6000\_TPR\_U15\_W | Unsigned | [32,8,15] | ZE(F<<2) | TPR(S) | Yes | R >> 2 |
| R\_C6000\_TPR\_U15\_D | Unsigned | [32,8,15] | ZE(F<<3) | TPR(S) | Yes | R >> 3 |
| R\_C6000\_TPR\_U32\_B | Unsigned | [32,0,326] | ZE(F) | TPR(S) | No | R |
| R\_C6000\_TPR\_U32\_H | Unsigned | [32,0,326] | ZE(F<<1) | TPR(S) | No | R >> 1 |
| R\_C6000\_TPR\_U32\_W | Unsigned | [32,0,326] | ZE(F<<2) | TPR(S) | No | R >> 2 |
| R\_C6000\_TPR\_U32\_D | Unsigned | [32,0,326] | ZE(F<<3) | TPR(S) | No | R >> 3 |
| R\_C6000\_SBR\_GOT\_U15\_W\_TLSMOD | Unsigned | [32,8,15] | ZE(F<<2) | GOT(TLSMOD(S)) + A - B | Yes | R >> 2 |
| R\_C6000\_SBR\_GOT\_U15\_W\_TBR | Unsigned | [32,8,15] | ZE(F<<2) | GOT(TBR(S)) + A - B | Yes | R >> 2 |
| R\_C6000\_SBR\_GOT\_U15\_W\_TPR\_B | Unsigned | [32,8,15] | ZE(F<<2) | GOT(TBR(S)) + A - B | Yes | R >> 2 |
| R\_C6000\_SBR\_GOT\_U15\_W\_TPR\_H | Unsigned | [32,8,15] | ZE(F<<2) | GOT(TBR(S)) + A - B | Yes | R >> 2 |
| R\_C6000\_SBR\_GOT\_U15\_W\_TPR\_W | Unsigned | [32,8,15] | ZE(F<<2) | GOT(TBR(S)) + A - B | Yes | R >> 2 |
| R\_C6000\_SBR\_GOT\_U15\_W\_TPR\_D | Unsigned | [32,8,15] | ZE(F<<2) | GOT(TBR(S)) + A - B | Yes | R >> 2 |
| R\_C6000\_SBR\_GOT\_L16\_W\_TLSMOD | Unsigned | [32,7,16] | ZE(F<<2) | GOT(TLSMOD(S)) + A - B | No | R >> 2 |
| R\_C6000\_SBR\_GOT\_L16\_W\_TBR | Unsigned | [32,7,16] | ZE(F<<2) | GOT(TBR(S)) + A - B | No | R >> 2 |
| R\_C6000\_SBR\_GOT\_L16\_W\_TPR\_B | Unsigned | [32,7,16] | ZE(F<<2) | GOT(TBR(S)) + A - B | No | R >> 2 |
| R\_C6000\_SBR\_GOT\_L16\_W\_TPR\_H | Unsigned | [32,7,16] | ZE(F<<2) | GOT(TBR(S)) + A - B | No | R >> 2 |
| R\_C6000\_SBR\_GOT\_L16\_W\_TPR\_W | Unsigned | [32,7,16] | ZE(F<<2) | GOT(TBR(S)) + A - B | No | R >> 2 |
| R\_C6000\_SBR\_GOT\_L16\_W\_TPR\_D | Unsigned | [32,7,16] | ZE(F<<2) | GOT(TBR(S)) + A - B | No | R >> 2 |
| R\_C6000\_SBR\_GOT\_H16\_W\_TLSMOD | Unsigned | [32,7,16] | ZE(F<<2) | GOT(TLSMOD(S)) + A - B | No | R >> 18 |
| R\_C6000\_SBR\_GOT\_H16\_W\_TBR | Unsigned | [32,7,16] | ZE(F<<2) | GOT(TBR(S)) + A - B | No | R >> 18 |
| R\_C6000\_SBR\_GOT\_H16\_W\_TPR\_B | Unsigned | [32,7,16] | ZE(F<<2) | GOT(TBR(S)) + A - B | No | R >> 18 |
| R\_C6000\_SBR\_GOT\_H16\_W\_TPR\_H | Unsigned | [32,7,16] | ZE(F<<2) | GOT(TBR(S)) + A - B | No | R >> 18 |
| R\_C6000\_SBR\_GOT\_H16\_W\_TPR\_W | Unsigned | [32,7,16] | ZE(F<<2) | GOT(TBR(S)) + A - B | No | R >> 18 |
| R\_C6000\_SBR\_GOT\_H16\_W\_TPR\_D | Unsigned | [32,7,16] | ZE(F<<2) | GOT(TBR(S)) + A - B | No | R >> 18 |
| R\_C6000\_TLSMOD | Unsigned | [32,0,32] | F | TLSMOD(S) | No | R |
| R\_C6000\_TBR\_U32 | Unsigned | [32,0,32] | F | TBR(S) | No | R |

# References

[1] The C6000 Embedded Application Binary Interface - Application Report, 2011, SPRAB89

[2] Executable and Linkable Format (ELF), [www.sco.com/developers/gabi/](http://www.sco.com/developers/gabi/)

[3] Ulrich Drepper, ELF Handling for Thread-Local Storage, [www.akkadia.org/**drepper**/tls.pdf](http://www.akkadia.org/drepper/tls.pdf), 2005, Version 0.20

[4] Alexandre Olivia and Glauber de Oliveira Costa, Speeding Up Thread-Local Storage Access in Dynamic Libraries in the ARM platform.