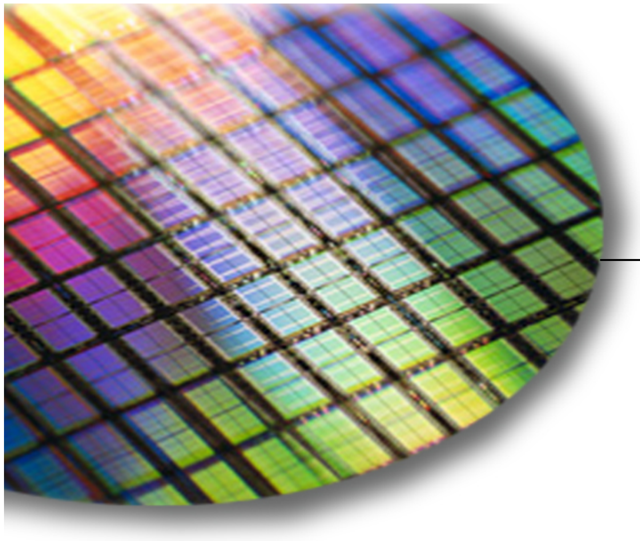


The World Leader in High-Performance Signal Processing Solutions

VDK: Core and Basic APIs

ANALOG DEVICES

Why do we need a kernel?

- A simple application that has to do only one task does not need an kernel.
  - It often just does the same thing over and over.
- If you have more than one task, an application could be structured in a few ways
  - Respond to external signals possibly exploiting a finite state machine to control the logic
  - Execute each of the tasks one after the other, doing high priority ones more often
- These approaches get difficult when
  - You need to preserve state to control what a sub-task does
  - Low priority tasks execute for longish times and delay high priority tasks

What does a kernel give you

- A headache and sometimes a nightmare
- Simplification of the preservation of state and the development of the control flow logic
- A structured way to control the relative priority of the different sub tasks
- Provides a development framework containing implementations of common synchronization and scheduling paradigms
- Efficient and thoroughly tested switching between the various tasks
- Support in understanding how you ended up in the mess you are in

What is VDK ?

- VDK is a kernel not an operating system
- VDK comprises:
  - VDK libraries
  - VDK specific ldf files
  - Include files
  - Template files
- Overheads
  - Memory overhead
  - Minimum memory requirement is platform dependent
  - Footprint is one of the most important metrics for a RT kernel
  - MIPS overhead

VDK Fundamentals

- **Threads**
  - **User code functionality is split between threads**
  - **Each thread has it's own stack**
- **Accessing shared resources**
  - **Used to synchronize activity**
  - **Semaphores, events, device flags, messages etc.**
- **Interrupts**
  - **Timer interrupt**
  - **Reschedule interrupt**

VDK Execution Environment

- After system startup, all code in a VDK application executes in one of levels:
  - Thread level
  - Kernel level
  - Interrupt level
- Trade-off between convenience and latency
  - The more functional the level, the longer it will take to respond to an external event

VDK Execution Environment

- After system startup, all code in a VDK application executes in one of levels:
  - Thread level
  - Kernel level
  - Interrupt level
- Trade-off between convenience and latency
  - The more functional the level, the longer it will take to respond to an external event

Kernel Level

- Lowest-priority (available) level
  - serviced by the “Reschedule ISR”
- Raised by software, scheduled by hardware
  - Masked by VDK e.g. during context switch
- Asynchronous wrt. Thread Level
- All pre-emptive rescheduling initiated from here
- C/C++ runtime environment
- Limited VDK API support
  - Functions must be interrupt-safe
- Device Driver “activate” functionality is the only user code which executes at this level
- ~500 cycle latency

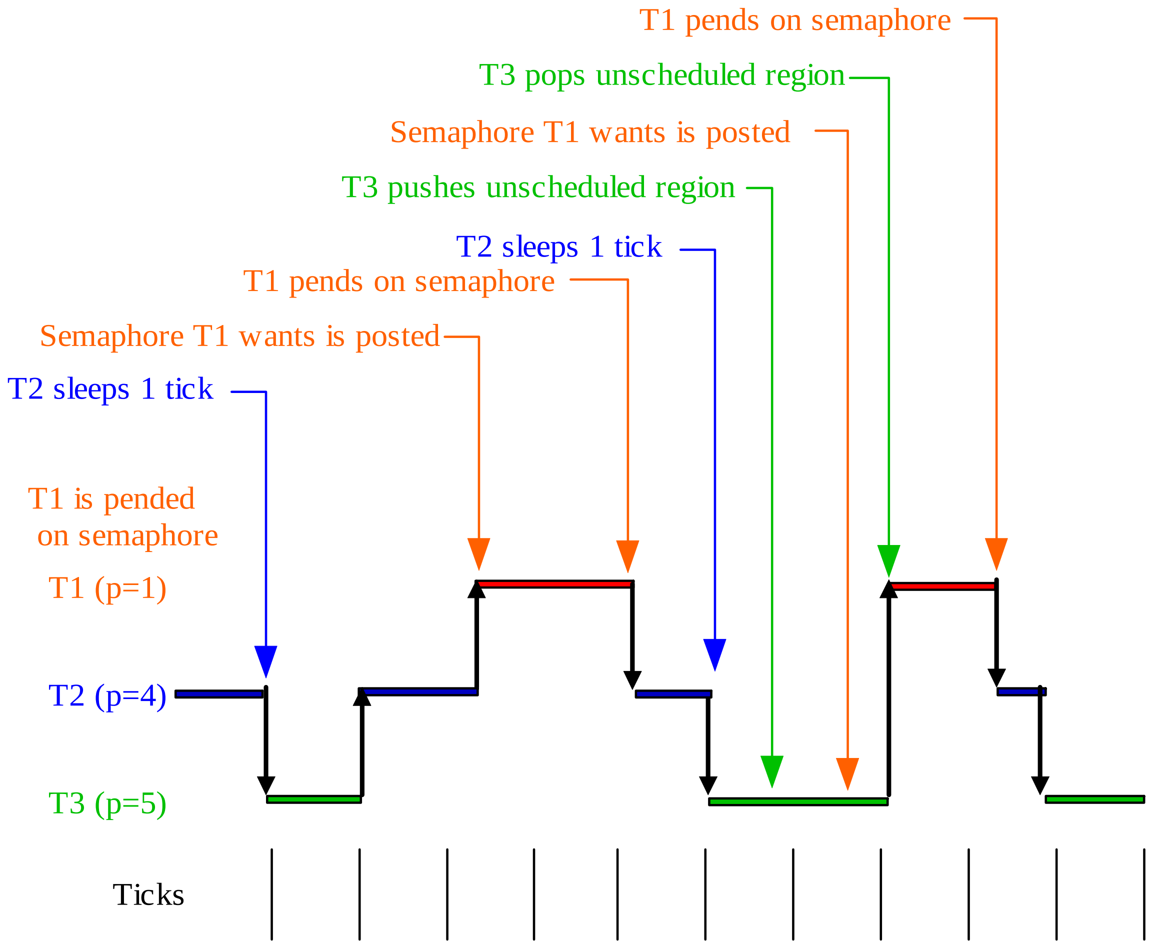
Threads

- Threads are instantiations of thread types
- Initial thread source and header files generated from templates
- Each thread has a unique ThreadID
- Each thread has it's own stack – allocated from the heap
- Overrunning a thread stack must be avoided
- Maximum stack usage can be determined
- No way to warn if a memory allocation request for a boot thread stack cannot be fulfilled

Thread Level

- Threads are scheduled in software, by the VDK Kernel
  - Co-operative and/or pre-emptive scheduling
- Runs in supervisor mode on TigerSHARC and BlackFIN
  - Reserves interrupt level 15 on BlackFIN
- All user thread code executes at this level
  - Also most VDK API code
- Full VDK API support
- Most other API functions supported
  - Functions must be thread-safe
- ~1000 cycle latency

Thread level scheduling



ANALOG DEVICES

Thread type generated source

- Each thread type has the following functions defined
  - An Init function (this does not execute in the new thread context)
  - A Run function that does the main work of the thread (usually a while loop)
  - An Error handler function that is invoked when VDK detects an error.
  - A Destroy function that is invoked as the thread instance dies.

Interrupt Level

- Collective term for all interrupts above Kernel level
- Raised by hardware, scheduled by hardware
  - Masked by VDK during critical activities
- Interrupt nesting supported
  - Interrupts must be explicitly enabled within ISR
- Asynchronous wrt both Thread level and Kernel level
- Written in assembly
  - C/C++ possible on Blackfin, with some extra work
- Very limited API support
  - Only ISR API macros supported by VDK
  - Any other functions called <u>must</u> be interrupt-safe
- ~100 cycle latency

Interrupts

- Source file for a user defined ISR generated from a template
- Any registers used by an ISR must be saved and restored first
- ISRs can be written in C, C++ or assembly (VDSP++ 4.0 onwards)
- Threads or device drivers can be triggered to allow use of high level code
- Interrupt masks should be accessed by VDK API calls
- ISR macros provided to:
  - Activate a device
  - Post a semaphore
  - Set/Clear an event bit

Managing Tasks in VDK

Scheduling

- Every thread has a priority level associated with it
- At any one time at most a single thread can be running
- Highest priority thread with all resource requirements fulfilled is the running thread
- If no user thread can run, the Idle thread is executed
- Scheduling the required thread can be effected by:
  - Using priorities
  - Resource requirements
  - Cooperation
  - Periodicity

Context Switching

- Reschedule ISR takes care of stopping the execution of one thread and starting the execution of another
- This context switch requires all appropriate registers to be saved/restored
- Speed of context switching is one of the most important metrics for a kernel

How to stop a thread from being switched out?

- **Unscheduled regions**
  - **Cannot change the running thread**
  - **Protects access of global variables**
  - **Allows multiple resource manipulations**
- **Critical regions**
  - **All interrupts are masked out**
  - **Protects access of global variables by ISRs**
  - **Interrupt latency is one of the most important metrics of a kernel**

VDK Error handling

- Errors or problems detected within an API function are not reported directly to the caller of the function
- Any errors are passed to the thread's error function
- The error handler can resolve some errors and return to the application normally
- Most errors cannot be recovered from however.
- The default error handler action is to terminate the thread.
- In style it is similar in structure to C++ exception handling.

ANALOG DEVICES

Inter-process communication

Semaphores

- All semaphores are "counting" in VDK 3.5
  - Use max. count of 1 for binary behaviour
- Interrupt level -> Thread level signaling
  - e.g. I/O completion
  - Counting behaviour can record multiple occurrences
- Thread -> Thread signaling
  - Mutual exclusion
    - But unscheduled regions may be more efficient
  - Resource counting
    - e.g. in parallel with a memory pool
- Can now be used from Kernel level
  - Restriction removed in 3.5

Messages

- Signals to synchronize thread activity
- Transfer information between threads
- Messages can be sent over a fixed number of channels
- Each channel is a FIFO
- Messages are received from channels in priority order
- Can pend on messages in a configurable manner

Messaging

- Thread -> Thread signaling (and data-transfer) only
- Provides a multi-wait capability
  - Channel priorities are only relevant when waiting on more than one channel
- Scheduling driven by data flow
- Messages can be forwarded or returned to sender
  - Recycling of messages and/or payloads may be more efficient than destruction
  - Returned messages can provide flow control
- “Ownership” of messages and payloads is important
- Payload management will be key to inter-processor messaging in future VDK

Message Payloads

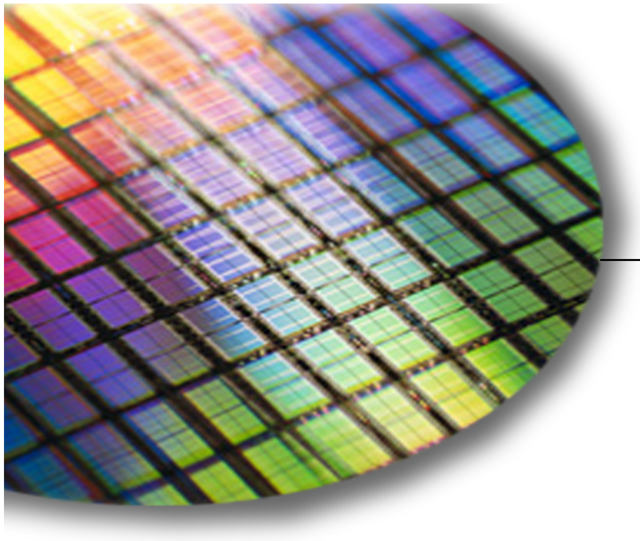
- **Each message carries three 32-bit items of information**
  - ***Type* is an integer, but is normally treated as an enumeration**
  - ***Size* is an unsigned integer**
  - ***Addr* (address) is a void ***
- **These attributes collectively define the message's payload**
- **Meaning of *Size* and *Addr* is programmer's choice**
  - **Interpretation is fixed for each valid value of *Type***
- **Payload can be carried:**
  - **Internally - in the 2x32 bits provided by *Size* and *Addr***
  - **Externally - in a data structure referenced by them**
- **VDK makes no interpretation of any part of the message payload**

Memory Pools

- Provides a block-based memory allocator
- Increased efficiency due to fixed size of blocks in each memory pool
- Prevents fragmentation
- Multiple pools can be defined with different block sizes
- Block construction at pool create or when used
- Messaging uses a memory pool

Events and Event bits

- **Signals used to synchronize thread activity**
- **Allow specification of multiple conditions**
- **Each event can be dependent on a user specified number of event bits**
- **Restriction on the number of events and event bits in a system**
- **Less efficient than semaphores**
- **When event is true then all threads pending on the event are unblocked**



The World Leader in High-Performance Signal Processing Solutions

VDK Device Drivers

What is a device driver?

- **Role of a device driver:**

“abstract the details of the hardware implementation from the software designer” –VDK manual VisualDSP++ 3.5

- **Note: In VisualDSP++ 3.5, device drivers are a part of the I/O interface. Device drivers are added to a VDK project as I/O objects. VisualDSP++ 2.0 device drivers are not compatible with VisualDSP++ 3.5 device drivers. See "Migrating Device Drivers" for a description of how to convert existing VisualDSP++ 2.0 device drivers for use in VisualDSP++ 3.5 projects.**

Device Driver: Dispatch function

- Only one interface to a device driver is through a dispatch function
- Dispatch function is called when the device is initialized, when a thread uses a device (open/close, read/write, control), or when an interrupt service routine transfers data to or from the device

I/O Interface and Device Drivers

- Device drivers are analogous to thread types
- A Boot I/O object is required to instantiate a device driver
- Dispatch function services:
  - Initialisation
  - Activation
  - Open
  - Close
  - SyncRead
  - SyncWrite
  - IOCtl

Only these 5 functions are available from the point of view of the thread

Device Flags

Device Flags

- Signals used to trigger thread activity
- Can be posted from ISRs
- Threads always block on device flags
- All blocked threads are released
- Always created dynamically (in the device driver Init function for example) using CreateDeviceFlag

DeviceDrivers and DeviceFlags

- **Driver activation:**
  - **No counting behaviour**
    - **Each driver can only occur once on activation queue**
  - **Interrupt Level -> Kernel Level signaling only**
- **DeviceFlags:**
  - **No counting behaviour**
    - **All pending threads released by post**
    - **A device flag self-resets on post**
  - **Kernel Level -> Thread Level signaling only**
  - **PushCriticalRegion() -> PendDeviceFlag() sequence is key to robust operation**
    - **Freeze state before deciding to block**

Device Flags

- Signals used to trigger thread activity
- Can be posted from ISRs
- Threads always block on device flags
- All blocked threads are released
- Always created dynamically (in the device driver Init function for example) using CreateDeviceFlag

Working with VDK

Creating a Project

- VDK support is added from the beginning
- A project must be structured/restructured to use VDK
- The IDDE generates 3 files for any VDK project:
  - The .vdk file stores the information entered into the kernel pane of the Project window
  - The vdk.cpp and vdk.h files contain the variable declarations and enumerations corresponding to the defined project
- All the various items are mapped to standard global variables and enums where the name is based on the user supplied name
  - Each thread type (such as Input) is mapped to an enum name such as kInput which acts as the thread identifier.

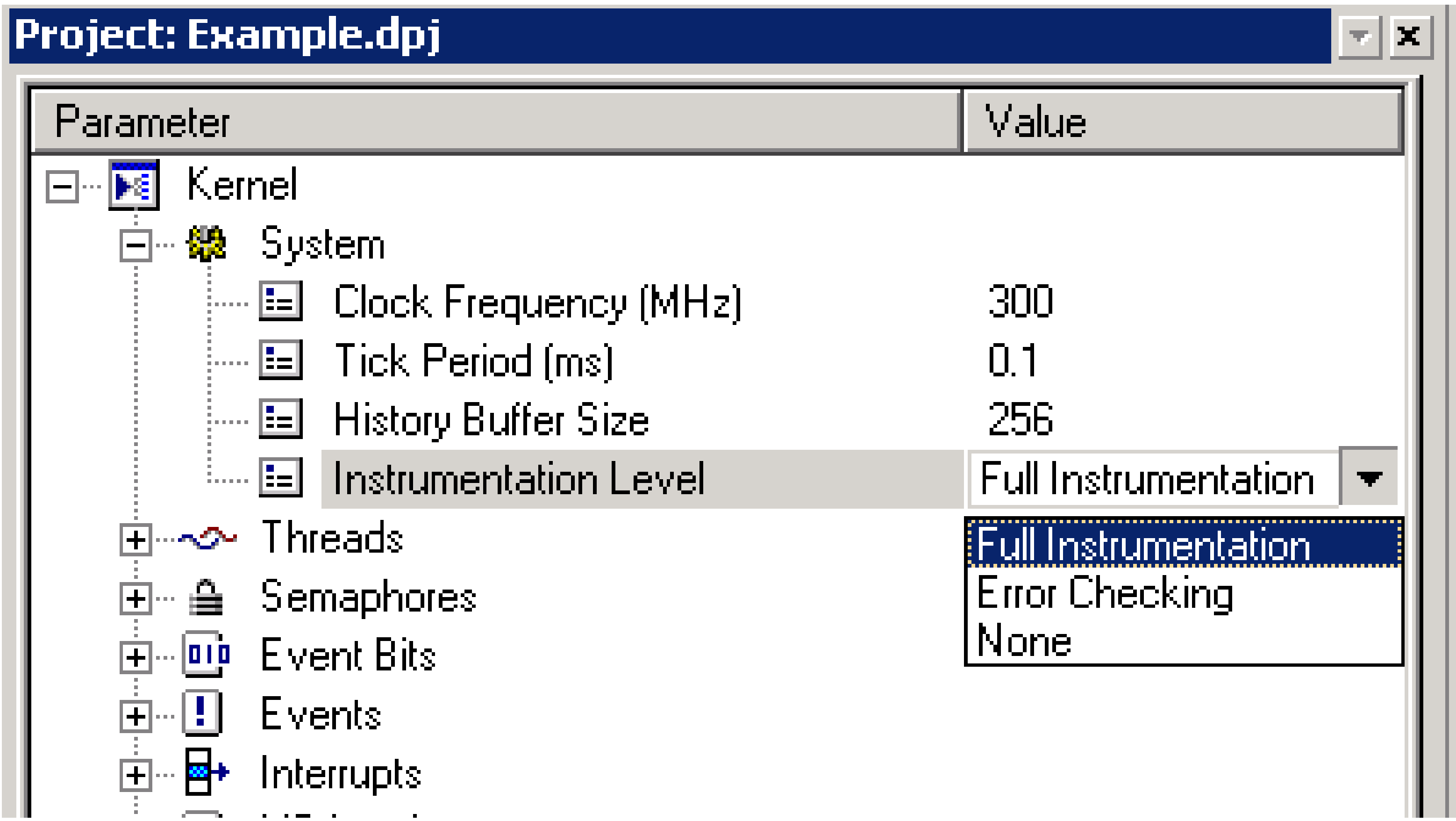
Generated files

- From information in the kernel tab the IDDE generates
  - Vdk.h and vdk.cpp which declares and defines the types and variables for items such as semaphores, messages etc
  - Vdk.h and vdk.cpp are updated when the kernel tab is updated
  - Vdk.h and vdk.cpp should not be updated directly
- Source files based on templates for
  - each thread type
  - each device driver
  - each interrupt that is defined
  - source file are not generated if a file of the same name already exists

The System Node

- Clock Frequency and Tick Period define number of cycles between VDK Ticks
- Each VDK Tick marked by a timer interrupt
- At least one timer interrupt reserved by VDK on each processor
- All time based services updated by the VDK Timer ISR
- Instrumentation Level defines level of debug support
- Full Instrumentation allows the use of the VDK State History window and provides Error Checking
- Error Checking provides additional sanity checks
- Instrumentation drastically increases code size
- History Buffer is wraparound, 4 words per entry

The System Node



API function names

- In C++ all of the VDK types and functions are defined within the VDK namespace
- In C++ an API function such as PopCriticalRegion is referred to as VDK::PopCriticalRegion
- In C the names are prefixed by VDK_
- In C PopCriticalRegion is referred to as VDK_PopCriticalRegion

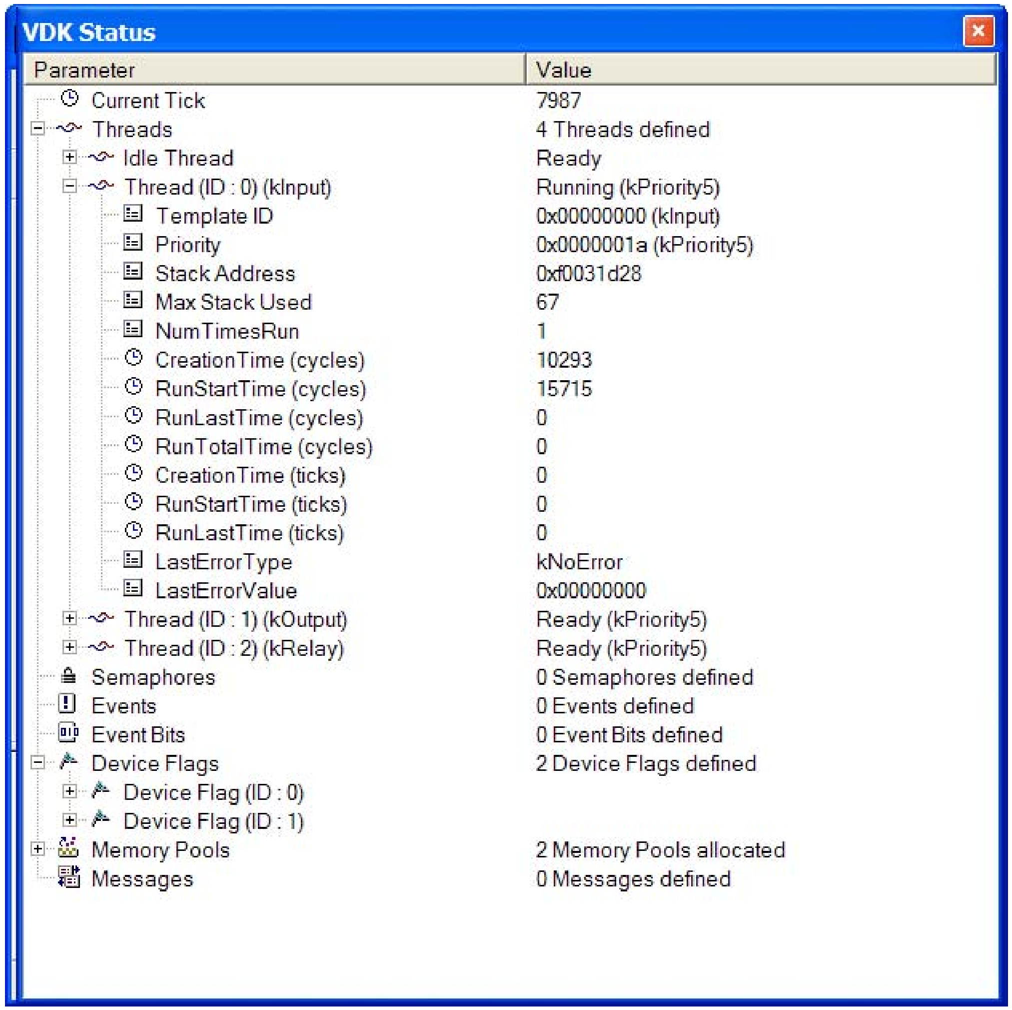
The ISR API

- Principally consists of these assembly macros (plus variations):
  - VDK_ISR_POST_SEMAPHORE_()
  - VDK_ISR_SET_EVENTBIT_()
  - VDK_ISR_CLEAR_EVENTBIT_()
  - VDK_ISR_ACTIVATE_DEVICE_()
- Only means of communication between an interrupt service routine and the VDK kernel.
- Mainly just change a small amount of internal state and raise the Reschedule interrupt. The Reschedule ISR may in turn:
  - action device activations
  - unblock waiting threads
  - perform a pre-emptive context switch

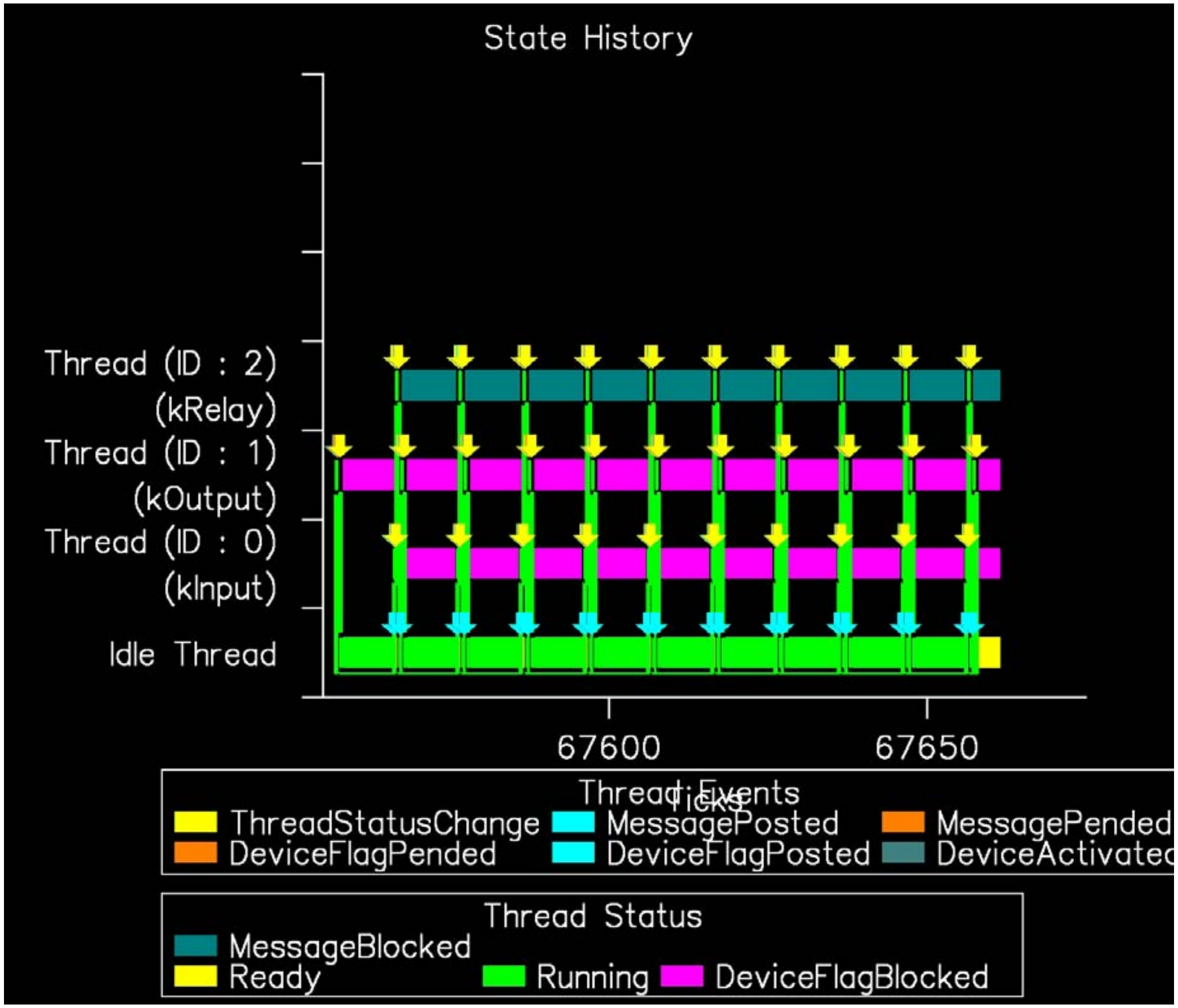
Debug assistance

- **VDK status window**
  - **State of each object**
  - **The current active thread**
  - **Which threads are waiting on what**
  - **Are threads waiting or ready to run**
- **VDK History window**
  - **Display the last set of events which have occurred**
  - **Helps to understand how you got where you are**

VDK Status window



VDK History window



VDK Core and Basic API Summary

- Provides a comprehensive set of services
- Is very efficient and at least as good as its competitors
- Provides the same functionality on the four families of processors
- Well integrated with the IDDE