LAUTERBACH
DEVELOPMENT TOOLS

API for Auxiliary Processing Unit

Release 09.2023

MANUAL

API for Auxiliary Processing Unit

API for Auxiliary Processing Unit

Version 10-Oct-2023

Introduction

This is the APU API Developer's Guide, intended for programmers developing a Sub Core Debugger. The API is also used to extend the built in disassembler and assembler for certain architectures (to support custom instructions).

It is not for someone who is just using the APU Debugger. Refer to **'APU'** in **"General Commands Reference Guide A"** (general_ref_a.pdf) instead.

In addition to the Main Core (CPU), some architectures optionally have one or more Sub Cores or Co-Processors, also called Auxiliary Processing Units (APUs) which are normally supported by LAUTERBACH. For example, most TriCore CPUs have the PCP as Sub Core, PowerPC and ColdFire may have one or two eTPUs and Star12X has the XGate. These Sub Cores are already supported by LAUTERBACH.

But for some reason the Sub Core may not be supported by the LAUTERBACH debugger although debugging is necessary. In these cases the CPU manufacturer or the user has the opportunity to write their own debugger which integrates into the LAUTERBACH debugging environment and benefits of its features.

Therefor the PowerView software contains an interface for integrating such an APU debugger via an external dynamically linked library, e.g. a Windows DLL or Linux sa. This PowerView interface is called APU API.

The APU API is built as a C/C++ library with a C function interface to the controlling application.

The term APU is a synonym for the Sub Core to be supported by the APU API.

Release Information

2007-01-02: On-chip breakpoints implemented, several bug fixes including HLL output in **APU.List** Window.

2006-11-26: Documentation started, first customers are already using the APU API.

Features

- Displaying code memory (disassembled and/ or HLL)
- Displaying data memory
- Accessing memory and registers (read and write)
- Go, Break and Single Step
- On-chip- and Software breakpoint support
- Address translation
- Loading symbols from object files
- Execution of PRACTICE commands in PowerView
- Defining own debugger commands

The libraries can be build for all platforms for which the Main Core Debugger is supported.

Requirements

The main requirement for an APU debugger is that there is already a debugger available for the Main Core. Consider you want to support the Data Mover of your preferred DSP, there must be a debugger for this DSP available from LAUTERBACH (for this DSP derivative to be exactly).

All APU debug registers and all APU memories have to be implemented in a memory accessible for the Main Core Debugger.

Infineon Cerberus IO Client

Some semiconductor companies have a standard debug port which is used in all of their chips.

As an example, Infineon uses the Cerberus as standard Debug Port for most of it's devices, e.g. XC16x, XC2000, TriCore and many others. So it is possible to address every memory and register via the system bus. In this special case the Cerberus IO Client can be regarded as the Main Core, and the effective core can be supported via the APU API.

For this, use the TriCore (32 bit architecture) or the C16x debugger (16 bit architecture) as Main Core Debugger. Select either Cerberus or Cerberus2 as CPU, this depends on the version of the Cerberus IO Client.

NOTE: Currently only 32 bit architectures are supported. Contact LAUTERBACH for more information.

Files

The APU API header- and implementation files and an example implementation can be found in the TRACE32 installation directory.

~~/demo/apu/t32apu.h	Header file.
~~/demo/apu/t32apu.c	Implementation for library.
~~/demo/apu/example_apu/	Example implementation for the Infineon PCP Coprocessor.
~~/demo/apu/example_apudisass/	Example for an APU implementing assembler and disassembler commands.
~~/demo/apu/example_virtualapu/	Example that can be used with any target or in an instruction set simulator. The subcore is simulated within the APU debugger.
~~/demo/apu/example_address_translation/	As above, but using a more complex setup with non-byte-addressed memories, address translation and user access.
~~/demo/apu/arm_corelink_dma_330_disass/	Functional disassembler for the Arm CoreLink 330 DMA controller.

Conventions

As convention, the following terms are being used:

- **Main Core**

  This is the host core or host core architecture which is required.

- **Sub Core** or **APU**

  The core where the new debugger is written for.

- **APU Debugger** or **Sub Core Debugger**

  The part of the debugger software within the library.

- **APU API**

  The interface between the APU Debugger and PowerView. PowerView is the software (GUI) with the Main Core Debugger provided by LAUTERBACH.

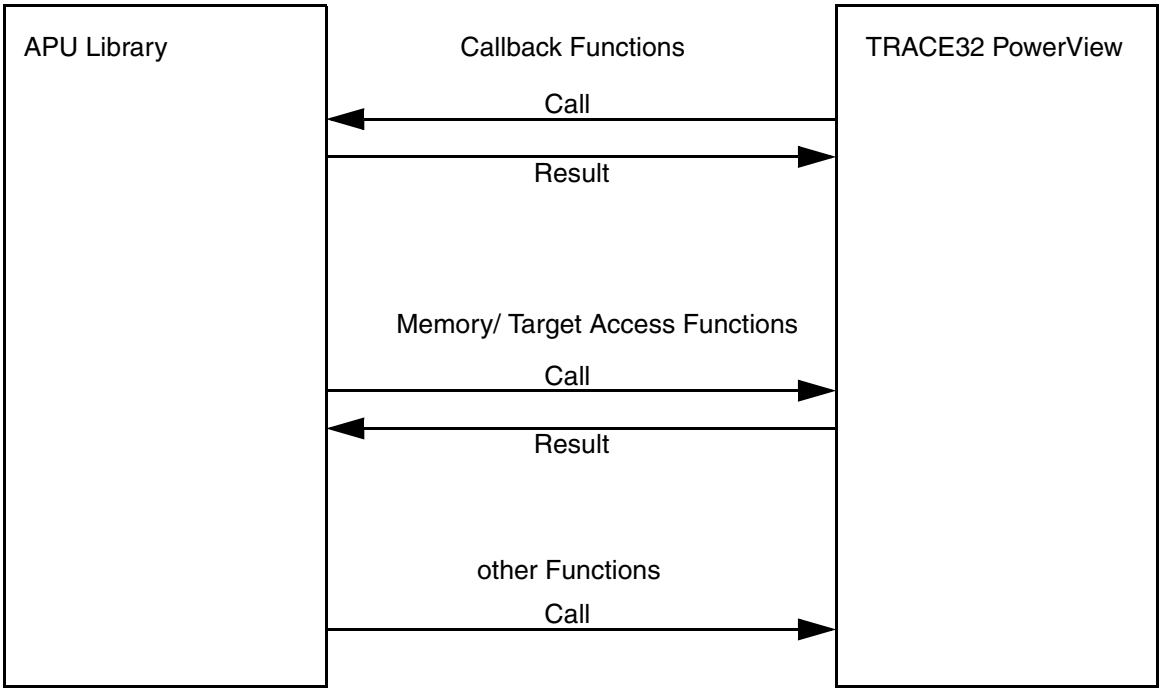
Basic Concept

All functionalities of the debugger are implemented in the external library which is linked to PowerView at runtime.

The user accesses the APU debugger by issuing the **APU** commands which are very similar to the well known debugger commands. E.g. for opening a **Data.List** window, the user issues **APU.List** instead.

Some of the low-level functionalities of each APU command is redirected to the APU library. E.g. for the **APU.List** window the disassembler integrated in the APU library is called for disassembling the opcodes. Data and debug information (e.g. about the Sub Core state) is stored within Callback Structures.

The library itself can make calls to the Main Core Debugger e.g. to read or write Main Core memory. It is also possible to execute arbitrary PRACTICE commands or to print error or warning messages.



Callback Functions

For implementing the debug features, PowerView has to perform APU specific code which is outsourced to the APU library. When needed, it is called by PowerView.

The following APU functionalities are implemented by Callback Functions:

- **SYStem.Up** of Main Core Debugger
- Get state (e.g. Running, Stopped or Idle)
- Step, Go and Break
- Memory read, write and translation
- Disassembler
- On-chip breakpoint set and delete
- Library specific commands
- APU unloading

Each function is registered by using the **Callback Register Functions**. Not all functionalities have to be implemented in the APU library. In this case they are not registered.

All Callback Functions have the same parameters passed:

apuContext context	Reserved for future use.
apuCallbackStruct *cbs	Callback structure for data and information exchange with PowerView.
apuPtr *proprietary	Pointer to a proprietary set of data.

`proprietary` can be used by the developer for any purpose.

All Callback Functions have the same return codes:

APU_OK	Callback Function exited successfully.
APU_FAIL	Callback Function failed.

Access to Main Core Debugger

In some cases the APU library needs access to some Main Core Debugger functions:

- Getting state of Main Core
- Accessing Main Core memory (read or write)
- Executing PowerView commands

See **Memory and Target Access Functions** for more information on how to use this functions.

Generic Configuration

PowerView needs to know information on the APU under debug:

- APU endianness for handling the byte order

  e.g. Litte Endian or Big Endian
- Memory Classes for implementing memory spaces

  e.g. Program and Data memory (Harvard Architecture)
- Software breakpoints

  to inform PowerView which Opcode implements the Software breakpoint

The APU endianness and at least Memory Class P (for Program Code) have to be defined.

See **Generic Configuration Functions** for more information on how to set up APU configuration for PowerView.

Output Functions

The **Generic Configuring Functions** also provide functions for status or error output. The output is directed to the **AREA** window.

Interface Functions

The Interface Functions provide the setup of the API after it is loaded.

Directly after loading, PowerView calls `APU_Interface()` for internal setup. From this, `APU_Init()` is called for APU specific setup. `APU_Init()` has to be provided by the APU specific code and is not included in the basic library functions.

`APU_Init()` implements the following tasks:

- Processing of optional API parameters
- Setting up APU specific parameters as APU endianness, Memory Classes and Software breakpoints
- Registering the Callback functions
- Sub Core specific setup (optional)

APU Callback Structures

The Callback Structures are used to pass information and data from PowerView to the Callback Functions and vice versa. Each Callback Function has its own Callback Structure defined which is passed by the **Global Callback Structure**:

```
typedef struct {
     int type;  /* type of referenced Callback Structure x */
     union {
          /* reference to Callback Structure */
          …
     } x;
} apuCallbackStruct;

/* Callback Structures referenced by global Callback Structure */
typedef struct { … } apuMemoryCallbackStruct;
typedef struct { … } apuGetstateCallbackStruct;
typedef struct { … } apuDisassemblerCallbackStruct;
…
```

APU Context

The APU context is reserved for future use.

APU Library

APU API Files

The API consists of one C source files and one C header file:

- **t32apu.h**

  This file contains the basic types and includes, and it handles the interface to the PowerView software.

- **t32apu.c**

  Handles the calls from PowerView and passes them to the user implemented functions.

Building the Library

Whenever a part of the application uses the API, the header file "`t32apu.h`" must be included. The corresponding C/C++ source file must contain the line

```
#include "t32.h"
```

quite at the beginning of the source.

When compiling and linking the application, the API files must be handled as normal source components of the application. Compilation could look like this:

```
cc -c t32apu.c

cc -c mydebugger.c
```

assuming, that the application is coded in a file called "`mydebugger.c`" and your C compiler is called "`cc`". The linker run is then invoked with

```
cc -o mydebugger t32apu.o mydebugger.o
```

assuming the linker name is "`cc`" and the object extension is "`o`".

Loading the Library

Set up PowerView as for debugging the main core.

Then the external library is loaded into PowerView:

```
APU.LOAD mydebugger.dll
```

Optionally any number of arguments may be passed to the DLL.

Writing a new Library

This section will help guide you through the first steps writing your first APU debugger library.

See also `demoapu.c` for an example.

The first function ever called is **APU_Init** on loading the APU Library. So this is the first function that has to be filled with life:

- Give the APU Debugger a name and add the compile date:

```
strcpy(cbs->x.init.modelname, __DATE__ "   APU Demo");
```

- Next, define the endianness:

```
APU_DefineEndianness(context, APU_ENDIANNESS_LITTLE);
```

- It is required to define at least Sub Core memory class `P`, but normally you should also define `D`:

```
APU_DefineMemory(context, 0, "D", 4, 4);
APU_DefineMemory(context, 1, "P", 2, 2);
```

In this example the access- and display width for `D` is 32 bit, the widths for `P` is 16 bit.

- As last step the mandatory Callback Functions have to be registered:

```
APU_RegisterGetStateCallback(context, GetStateDemo, 0);
APU_RegisterMemoryReadCallback(context, ReadDemoMemory, 0);
APU_RegisterMemoryWriteCallback(context, WriteDemoMemory, 0);
APU_RegisterDisassemblerCallback(context, DisassembleDemo, 0, 2, 4);
APU_RegisterGoCallback(context, GoDemo, 0);
APU_RegisterBreakCallback(context, BreakDemo, 0);
APU_RegisterStepCallback(context, StepDemo, 0);
```

Here the disassembler registers with a minimum instruction length of 2 and a maximum instruction length of 4.

Note that the **Translate Callback Function** is required for some basic functions as breakpoints and should be implemented as soon as possible.

Implementation of the Callback Functions

The Callback functions are now registered but not yet existing. As first step it is required to create every Callback Function that is registered with a minimum (empty) functionality.

Now the Callback Functions have to be filled with life. There are several concepts how to do this, but the most important tasks are:

- **Determining the Sub Core state (GetState Callback Function)**

  When the Sub Core is running, the memory access Callback Functions are never called (no matter of the Main Core state). So it is not possible to perform any other action. If it is difficult to determine the Sub Core state at this point of time, a temporary workaround may be to return `APU_STATE_STOPPED` all the time.

- **Implementing Sub Core data display**

  As soon as possible the memory read- and write Callback Functions will have to be implemented. Although more advanced features as memory mapping or MMU support may be omitted at this point of time, it is strongly required to test this functionality very intensive.

The next steps might be to implement the basic debugging features:

- **Disassembler Callback Function**
- **Step**, **Go** and **Break Callback Functions**
- Software- and on-chip breakpoints

Fine Tuning

The fine tuning usually is the last step, but you may want or need to do some parts at an earlier stage.

- Initializing the Sub Core on **System.Mode Up** of the Main Core (**Reset Callback Function**)
- Recognition of special Sub Core states
- **Exit Callback Function** for safe exit on unload

Symbol Information

In most cases the code for the Sub Core is included in the Main Core object file. The Main Core sets up the Sub Core and loads code and data to its memory.

In most cases the object file maps the symbol information to the address space of the Sub Core. The **Translate Callback Function** defines how the Sub Core address space is mapped to the Main Core address space, so TRACE32 PowerView can map the symbol information from Main Core to Sub Core and vice versa.

However it might be necessary to provide manually created symbol information. In this case the **Local Symbol Files** from **"General Commands Reference Guide D"** (general_ref_d.pdf) might be a suitable solution.

APU Commands

This is just an overview of the most important APU commands. See **'APU'** in **"General Commands Reference Guide A"** (general_ref_a.pdf) for detailed information.

APU.Break.direct	Program break
APU.Break.Set	Set breakpoint
APU.command	Send command to APU library
APU.Data.dump	Memory dump
APU.Go	Real-time execution
APU.List	Symbolic display
APU.LOAD	Load APU library
APU.Register	Show APU register window
APU.RESet	Reset APU core
APU.Step	Single-stepping
APU.View	Display peripherals

APU Entry Functions

APU_Interface

```
Prototype:      int APUAPI APU_Interface(
                     apuContext context,
                     apuCallbackStruct *cbs
                )
```

context	APU context.
***cbs**	Callback structure.

This function is called once from PowerView directly after loading the APU library. The callback structure contains a pointer to the arguments passed with **APU.LOAD**.

This function is provided by LAUTERBACH and normally there is no need to make any changes.

APU_Init() is called at the end of this function.

APU_Init

```
Prototype:      int APUAPI APU_Init(
                     apuContext context,
                     apuCallbackStruct *cbs
                )
```

context	APU context.
***cbs**	Callback structure.

This function is called from **APU_Interface()**. This function is not supplied by LAUTERBACH and has to be implemented by the developer for registering the callback functions and setting up the APU debugger.

APU_DefineEndianness

Prototype:

```
void APUAPI APU_DefineEndianness(
     apuContext context,
     int endianness
)
```

context	APU context.
endianness	Define the target endianness.

PowerView needs to know how to assemble the bytes for non-8 bit values. The following endiannesses are possible:

APU_ENIDANESS_LITTLE Little Endian mode.

APU_ENIDANESS_BIG Big Endian mode.

APU_DefineMemory

Prototype:

```
void APUAPI APU_DefineMemory(
     apuContext context,
     int id,
     const char *name,
     int width,
     int flags
)
```

context	APU context.
id	ID of memory class. IDs should start at 0 and should be continuous. The maximum allowable ID is 15.
***name**	Name of memory class. Names should consist of 1 to 3 letters, no numbers allowed.
width	Memory access width in bytes.
flags	Default display width in bytes.

Define the memory classes.

Each memory class represents either a logical or physical independent memory and has it's own ID. At least memory class P for code with `id=1` is mandatory.

The access width tells the Main Core Debugger which access width to use for accessing the target memory. There may be restrictions for certain memory regions. The access width also defines the address unit that is exposed to the user. Note that addresses passed through the APU API are always byte addresses.

The display width defines the default width for displaying data in a window.

The following names are used by the debugger:

"D"	Used for accesses to the D: access class for APU-related operations.
"P"	Used for accesses to the P: access class for APU-related operations.
"USR"	Used for accesses to the USR: access class, even for commands that do not normally access the APU. If a loaded APU defines this memory, it has precedence over a **Data.USRACCESS** external access algorithm.

APU_DefineSoftbreak (optional)

Prototype:

```
void APUAPI APU_DefineSoftbreak(
     apuContext context,
     int width,
     const unsigned char *data
)
```

context	APU context.
width	Software breakpoint opcode width (in bytes).
***data**	Software breakpoint opcode.

Define the opcode used for a software breakpoint.

APU_Printf

Prototype:

```
void APUAPI APU_Printf(
     apuContext context,
     const char *format,
     …
)
```

context	APU context.
***format, …**	`printf()` compliant format string.

Print an information message (e.g. status message) into the PowerView **AREA** window.

APU_Warning

Prototype:

```
void APUAPI APU_Warning(
     apuContext context,
     const char *format,
     …
)
```

context	APU context.
***format, …**	`printf()` compliant format string.

Print a warning message into the PowerView **AREA** window.

APU_GetSymbol

Prototype:

```
int APUAPI APU_GetSymbol(
     apuContext context,
     const char *name,
     apuWord * paddress,
     int * pflags
)
```

context	APU context.
***name**	Symbol name that needs to be resolved.
***paddress**	Address of the symbol.
***pflags**	Extra address flags.

Resolve a symbolic name. The return value is APU_OK when the name could be resolved. The address is returned via padddress and pflags.

Callback Register Functions

APU_RegisterBreakCallback

Prototype:

```
void *APUAPI APU_RegisterBreakCallback(
     apuContext context,
     apuCallbackFunctionPtr func,
     apuPtr proprietary
)
```

context	APU context.
func	Function which handles the callback.
proprietary	Pointer to proprietary data structure.

Register the function `func` as Break Callback Function which is called from the **APU.Break** command.

When called the Callback Function stops the real-time execution of the APU.

APU_RegisterBreakpointCallback (optional)

Prototype:

```
void *APUAPI APU_RegisterBreakpointCallback(
     apuContext context,
     apuCallbackFunctionPtr func,
     apuPtr proprietary,
     int bptypes
)
```

context	APU context.
func	Function which handles the callback.
proprietary	Pointer to proprietary data structure.
bptypes	Breakpoint types available for the Sub Core.

Register the function `func` as Breakpoint Callback Function for handling on-chip breakpoints.

`bptypes` tells PowerView what types of on-chip breakpoints are available for the Sub Core. It is currently not possible to set on-chip breakpoints on read or write data.

APU_BPTYPE_PROGRAM	Code breakpoints.
APU_BPTYPE_READ	Data breakpoints on read access (address).
APU_BPTYPE_WRITE	Data breakpoints on write access (address).

APU_RegisterCommandCallback (optional)

Prototype:

```
void *APUAPI APU_RegisterCommandCallback(
     apuContext context,
     apuCallbackFunctionPtr func,
     apuPtr proprietary
)
```

context	APU context.
func	Function which handles the callback.
proprietary	Pointer to proprietary data structure.

Register the function `func` as Command Callback Function for the **APU.command** command.

When **APU.command** is called in PowerView, all its parameters are passed to the Command Callback Function which processes the parameters. Usually this is used to extend the APU API with extra commands and options (e.g. APU.SYStem.Options). Note that all parameters are parsed by the PowerView command line parser for formal correctness.

APU_RegisterDisassemblerCallback

Prototype:

```
void *APUAPI APU_RegisterDisassemblerCallback(
     apuContext context,
     apuCallbackFunctionPtr func,
     apuPtr proprietary,
     int mininstlen,
     int maxinstlen,
)
```

context	APU context.
func	Function which handles the callback.
proprietary	Pointer to proprietary data structure.
mininstlen	Minimum instruction length (in bytes) an APU opcode can have.
maxinstlen	Maximum instruction length (in bytes) an APU opcode can have.

Register the function `func` as Disassembler Callback Function which is called from the **Data.List** window for decoding opcodes. This API can also be used by some architectures to implement a disassembler for custom instructions.

APU_RegisterAssemblerCallback

Prototype:

```
void *APUAPI APU_RegisterAssemblerCallback(
     apuContext context,
     apuCallbackFunctionPtr func,
     apuPtr proprietary
)
```

context	APU context.
func	Function which handles the callback.
proprietary	Pointer to proprietary data structure.

Register the function `func` as Assembler Callback Function which is called from the **Data.Assemble** command for coding mnemonics. This API can also be used by some architectures to implement an assembler for custom instructions.

APU_RegisterExitCallback (optional)

Prototype:

```
void *APUAPI APU_RegisterResetCallback(
     apuContext context,
     apuCallbackFunctionPtr func,
     apuPtr proprietary
)
```

context	APU context.
func	Function which handles the callback.
proprietary	Pointer to proprietary data structure.

Register the function `func` as Exit Callback Function which is called from the **APU.RESet** command.

This gives the APU library the opportunity to quit safely: Close all files, free allocated memory, unload other libraries, … .

APU_RegisterGetStateCallback

Prototype:

```
void *APUAPI APU_RegisterGetStateCallback(
     apuContext context,
     apuCallbackFunctionPtr func,
     apuPtr proprietary
)
```

context	APU context.
func	Function which handles the callback.
proprietary	Pointer to proprietary data structure.

Register the function `func` as GetState Callback Function for evaluating the current state of the APU: Running, stopped or idle. In case the state is stopped, the current PC is also read.

Note that the GetState Callback function is continuously called by PowerView (for changing the interval see **SETUP.URATE**).

APU_RegisterGoCallback

Prototype:

```
void *APUAPI APU_RegisterGoCallback(
     apuContext context,
     apuCallbackFunctionPtr func,
     apuPtr proprietary
)
```

context	APU context.
func	Function which handles the callback.
proprietary	Pointer to proprietary data structure.

Register the function `func` as Go Callback Function which is called from the **APU.Go** command.

When called the function starts the real-time execution of the APU.

APU_RegisterMemoryReadCallback

Prototype:

```
void *APUAPI APU_RegisterMemoryReadCallback(
     apuContext context,
     apuCallbackFunctionPtr func,
     apuPtr proprietary
)
```

context	APU context.
func	Function which handles the callback.
proprietary	Pointer to proprietary data structure.

Register the function `func` as Read Memory Callback Function which is used for reading APU memory.

The memory read function is also responsible for any memory mapping and/or MMU functionality as long as the Sub Core implements such features.

APU_RegisterMemoryWriteCallback

```
Prototype:      void *APUAPI APU_RegisterMemoryWriteCallback(
                     apuContext context,
                     apuCallbackFunctionPtr func,
                     apuPtr proprietary
                )
```

context	APU context.
func	Function which handles the callback.
proprietary	Pointer to proprietary data structure.

Register the function `func` as Write Memory Callback Function which is used for writing APU memory.

The memory read function is also responsible for any memory mapping and/or MMU functionality as long as the Sub Core implements such features.

APU_RegisterResetCallback (optional)

```
Prototype:      void *APUAPI APU_RegisterResetCallback(
                     apuContext context,
                     apuCallbackFunctionPtr func,
                     apuPtr proprietary
                )
```

context	APU context.
func	Function which handles the callback.
proprietary	Pointer to proprietary data structure.

Register the function `func` as Reset Callback Function which is called after every **SYStem.Up** in the Main Core Debugger. This means that the Main Core has been taken out of reset and was initialized.

The APU library can now initialize and set up the Sub Core for debugging. Note that for **SYStem.Mode Down** no Callback Function is called.

APU_RegisterStepCallback

Prototype:

```
void *APUAPI APU_RegisterStepCallback(
     apuContext context,
     apuCallbackFunctionPtr func,
     apuPtr proprietary
)
```

context	APU context.
func	Function which handles the callback.
proprietary	Pointer to proprietary data structure.

Register the function `func` as Step Callback Function which is called from the **APU.Step** command.

When called the function performs a single step in assembler mode.

APU_RegisterTranslateCallback (optional)

Prototype:

```
void *APUAPI APU_RegisterTranslateCallback(
     apuContext context,
     apuCallbackFunctionPtr func,
     apuPtr proprietary
)
```

context	APU context.
func	Function which handles the callback.
proprietary	Pointer to proprietary data structure.

Register the function `func` as Translate Callback Function.

The Translate Callback is called when PowerView needs to know which Sub Core memory address corresponds to which Main Core memory address and vice versa. It is mainly needed for correct symbol and HLL displaying.

APU_GetState

Prototype:

```
void *APUAPI APU_GetState(
     apuContext context,
     int *pstate
)
```

context	APU context.
***pstate**	Main Core target state.

Get the state of the Main Core.

It can be in one of the following values states:

APU_STATE_STOPPED The Main Core is not executing code.

APU_STATE_RUNNING The Main Core is currently executing code.

APU_STATE_IDLE The Main Core is in idle state.

Prototype:

```
void *APUAPI APU_ReadMemory(
     apuContext context,
     apuWord address,
     int flags,
     unsigned char *pdata,
     int size,
     int width
)
```

context	APU context.
address	Start address to read from.
flags	Main Core memory class to read from.
***pdata**	Buffer for read data in APU endianness.
size	Length of read data in bytes.
width	Access width in bytes.

Read memory from the Main Core address space.

The following memory classes are valid for all main core architectures:

T32_MEMORY_ACCESS_DATA	Logical data memory.
T32_MEMORY_ACCESS_PROGRAM	Logical program memory.
T32_MEMORY_ACCESS_USR	User-specified memory (normally not required).
T32_MEMORY_ACCESS_VM	Virtual memory (normally not required).
T32_MEMORY_ATTR_DUALPORT	Modifier for run-time access.

Access to virtual memory or user memory is only required in very rare cases.

APU_WriteMemory

Prototype:

```
void *APUAPI APU_WriteMemory(
     apuContext context,
     apuWord address,
     int flags,
     unsigned char *pdata,
     int size,
     int width
)
```

context	APU context.
address	Start address to write to.
flags	Main Core memory class to write to.
***pdata**	Buffer for write data in APU endianness.
size	Length of write data in bytes.
width	Access width in bytes.

Write memory via Main Core Debugger.

See **APU_ReadMemory** for more information.

APU_ExecuteCommand

Prototype:

```
void *APUAPI APU_ExecuteCommand(
     apuContext context,
     char *cmdline
)
```

context	APU context.
***cmdline**	Command to execute from PowerView.

Execute command within PowerView.

This can be used to obtain information from the Main Core Debugger or to control it.

Breakpoint Callback Structure

Prototype:

```
typedef struct {
     apuWord address;
     apuWord addressto;
     int flags;
     int bptype;
     int bpid;
} apuBreakpointCallbackStruct;
```

context	APU context.
address	Breakpoint start address.
addressto	Breakpoint end address.
flags	Memory class.
bptype	Type of breakpoint (program, data read or data write).
bpid	Breakpoint ID.

PowerView supports on-chip breakpoints either on a single address or on an address range. In case of a single address, `address` and `addressto` are equal. If the Sub Core does not support on-chip breakpoints on address ranges, the APU Debugger has to shrink down the range to a single address.

The APU API currently supports the following on-chip breakpoint types, which may also be combined. There is no support for on-chip breakpoints on data values.

APU_BPTYPE_PROGRAM	Program breakpoints in code
APU_BPTYPE_READ	Data breakpoint on read access (data address)
APU_BPTYPE_WRITE	Data breakpoint on write access (data address)

When a new on-chip breakpoint is requested to be set, `bpid` is set to `0` by PowerView. The APU Debugger checks if the on-chip breakpoint can be set. If so, the on-chip breakpoint is programmed and `bpid > 0` is assigned and returned to PowerView. When an on-chip breakpoint is requested to be deleted, the respective `bpid` is passed.

If an on-chip breakpoint can not be set this is indicated with `bpid=0`.

Note that it is currently not possible to use software breakpoints and on-chip breakpoints for code simultaneously.

Global Callback Structure

Prototype:

```
typedef struct {
     int type;
     union {
          apuParamCallbackStruct init;
          apuParamCallbackStruct command;
          apuMemoryCallbackStruct memory;
          apuDisassemblerCallbackStruct dis;
          apuAssemblerCallbackStruct ass;
          apuGetstateCallbackStruct state;
          apuTranslateCallbackStruct translate;
          apuBreakpointCallbackStruct breakpoint;
     } x;
} apuCallbackStruct;
```

type	Callback type.
x	Pointer to the Callback Structure of the current Callback Function.

The Unified Callback Structure is passed to every Callback Function when called. Any information or data to be exchanged between PowerView and the Callback Function is placed here.

`type` can have the following values:

Value of `type`	Valid member of `x`	Registered using
APU_CALLBACK_BREAK	*none*	APU_RegisterBreakCallback()
APU_CALLBACK_BREAKPOINT	breakpoint	APU_RegisterBreakpointCallback()
APU_CALLBACK_COMMAND	command	APU_RegisterCommandCallback()
APU_CALLBACK_DISASSEMBLER	dis	APU_RegisterDisassemblerCallback()
APU_CALLBACK_ASSEMBLER	ass	APU_RegisterAssemblerCallback()
APU_CALLBACK_EXIT	*none*	APU_RegisterExitCallback()
APU_CALLBACK_GETSTATE	state	APU_RegisterGetStateCallback()
APU_CALLBACK_GO	*none*	APU_RegisterGoCallback()
APU_CALLBACK_INIT	init	*used for APU_Init() / APU_Interface()*
APU_CALLBACK_MEMORYREAD	memory	APU_RegisterMemoryReadCallback()
APU_CALLBACK_MEMORYWRITE	memory	APU_RegisterWriteCallback()
APU_CALLBACK_RESET	*none*	APU_RegisterResetCallback()
APU_CALLBACK_STEP	*none*	APU_RegisterStepCallback()
APU_CALLBACK_TRANSLATE	translate	APU_RegisterTranslateCallback()

Disassembler Callback Structure

Prototype:

```
typedef struct {
     apuWord address;
     int flags;
     unsigned char *data;
     char *mnemo;
     char *comment;
     int instlen;
     int jumpflag;
     apuWord jumptarget;
} apuDisassemblerCallbackStruct;
```

address	Address of instruction in memory.
flags	Memory class of instruction.
***data**	Memory content starting at given address.
***mnemo**	Disassembled mnemonic.
***comment**	Optional comment (not part of the mnemonic).
instlen	Length of decoded instruction in bytes.
jumpflag	Type of jump.
jumptarget	Specifies the jump target if direct jump instruction.

The Disassembler Callback function obtains the address and the memory class of the opcode to disassemble. Additionally the opcode at this address is passed.

The disassembler returns the decoded mnemonic of the opcode and optionally a comment. It also returns the length of the decoded instruction in `instlength`. This is especially necessary for Sub Cores with variable instruction length.

Only in case the decoded instruction is a jump, the `jumpflag` has to be set to indicate the type of the jump. If the jump is a direct jump, `jumptarget` has to be provided. The following table shows the possible jump flags:

APU_JMPFLG_DIRECT	Unconditional jump with direct target.
APU_JMPFLG_DIRECTCOND	Conditional jump with direct target.
APU_JMPFLG_INDIRECT	Unconditional jump with indirect target.
APU_JMPFLG_INDIRECTCOND	Conditional jump with indirect target.

If the instruction was disassembled successfully, the callback returns with APU_OK, otherwise (e.g. undefined opcodes) with APU_FAIL.

Assembler Callback Structure

Prototype:

```
typedef struct {
     apuWord address;
     int flags;
     int complete;
     const char * mnemo;
     unsigned char *data;
     int instlen;
     int mnemolen;
     char * errormessage;
     int errorpos;
} apuAssemblerCallbackStruct;
```

address	Address of instruction in memory.
flags	Memory class of instruction.
complete	Set when the input string is complete (otherwise the command is being typed).
***mnemo**	Input string for the assembler.
***data**	Resulting code from the assembler.
instlen	Length of coded instruction in bytes (or zero if this mnemonic is not handled).
mnemolen	Length of the mnemonic (returned by the assembler). A new instruction may start here.
***errormessage**	Error message. Set when the assember wishes to throw an error message. The error message should begin with “!#” to be treated as a text message.
errorpos	Relative position of the error in the mnemo string.

The Assembler Callback function obtains the address and the memory class of the place to assemble. Additionally the command line and information if the command line is complete is passed.

The assembler returns the coded opcodes and length and the length of the input string that was parsed. In error cases it can produce an error message for the user that points directly to the error position. Returning without the mnemolen field set means that the assembler was NOT processing the mnemonics and the “regular” assembler should try parsing it.

If the instruction was assembled successfully, the callback returns with `APU_OK`, otherwise (e.g. undefined mnemonics) with `APU_FAIL`.

Prototype:

```
typedef struct {
     apuWord pc;
     int state;
     char *text;
} apuGetstateCallbackStruct;
```

pc	Current program counter.
state	Current Sub Core state.
***text**	Special state.

The Sub Core state can have the following values:

APU_STATE_STOPPED The Sub Core is not executing code,

APU_STATE_RUNNING The Sub Core is currently executing code.

APU_STATE_IDLE The Sub Core is in Idle state.

In case Program Counter can not be read when the target is running is does not need to be provided.

In case the target is running in a special state (e.g. Power Save, Suspend, …), the string provided via `text` is shown in the status field next to running. `state` must be `APU_STATE_RUNNING` for this.

Memory Callback Structure

Prototype:

```
typedef struct {
     apuWord address;
     int flags;
     int length;
     int width;
     unsigned char *data;
} apuMemoryCallbackStruct;
```

address	Start address.
flags	Sub Core memory class.
length	Access length in bytes.
width	Access width in bytes.
***data**	Pointer to container for read or write data.

The Memory Callback Structure is the same for read and write accesses.

Prototype:

```
typedef struct {
    int     version;
    char  * modelname;
    char  * commandline;
    int     argc;
    char ** argp;
    int   * argpint;
    apuWord   * argpword;
    apuWord32 * argpword32;
    apuWord64 * argpword64;
    apuWord   * argpaddress;
    apuWord32 * argpaddress32;
    apuWord64 * argpaddress64;
    apuWord   * argpwordupper;
    apuWord32 * argpwordupper32;
    apuWord64 * argpwordupper64;
    apuWord   * argpaddressupper;
    apuWord32 * argpaddressupper32;
    apuWord64 * argpaddressupper64;
    char ** argpstring;
    int   * argptype;
} apuParamCallbackStruct;
```

version	APU version of TRACE32 executable.
***modelname**	Name of the Sub Core Debugger.
***commandline**	Pointer to **APU.LOAD** command and all arguments.
argc	Number of arguments passed to library.
argp*	See below

The version information is set by the APU API. `modelname` is the name of the Sub Core Debugger and is chosen by the developer. It is recommended to add the compile time also.

The first argument to APU.command is interpreted as a keyword. All remaining arguments are interpreted as expressions. Type and result of the expression is passed in the **argp*** fields. If the result is integral (**argptype[i]** is one of APU_PARAM_TYPE_BOOL, APU_PARAM_TYPE_INT, APU_PARAM_TYPE_ADDRESS), the fields **argpint[i]**, **argpword[i]**, **argpword32[i]**, **argpword64[i]**, **argpaddress[i]**, **argpaddress32[i]** and **argpaddress64[i]** all contain the same value, possibly truncated if the value is too big for the respective type. For ranges (APU_PARAM_TYPE_INTRANGE, APU_PARAM_TYPE_ADDRESSRANGE), **argpwordupper32[i]**, **argpwordupper64[i]**, **argpaddressupper[i]**, **argpaddressupper32[i]** and **argpaddressupper64[i]** all contain the upper noninclusive bound of the range. For APU_PARAM_TYPE_STRING, **argpstring[i]** contains the result of the expression.

For any type, **argp[i]** contains the expression that was given by the user before evaluation.

Note that **argpstring** and **argptype** are only available if **version** is at least 2.

For an example of how to interpret the command line arguments, see DumpParameters() in ~~/demo/apu/example_virtualapu/example_virtualapu.c.

Translate Callback Structure

Prototype:

```
typedef struct {
     apuWord address;
     int flags;
     int direction;
} apuTranslateCallbackStruct;
```

address	Address to translate.
flags	Memory Class.
direction	Translate direction.

The direction flag indicates the direction of the address translation:

APU_TRANSLATE_TO_MAINCORE Translate from Sub Core to Main Core address space.

APU_TRANSLATE_TO_SUBCORE Translate from Main Core to Sub Core address space.

Version Control

Document version control:

Version	Date	Change
0.1	2007-01-08	More description added, on-chip breakpoints added.
0.0	2006-11-08	Documentation started.