Das neue (interne) GenericDriver Konzept

Draft

V0.91

Jesko Schwarzer, Systemberatung (2019-März/April)

in Kooperation mit Universität zu Köln und Fraunhofer Gesellschaft St. Augustin

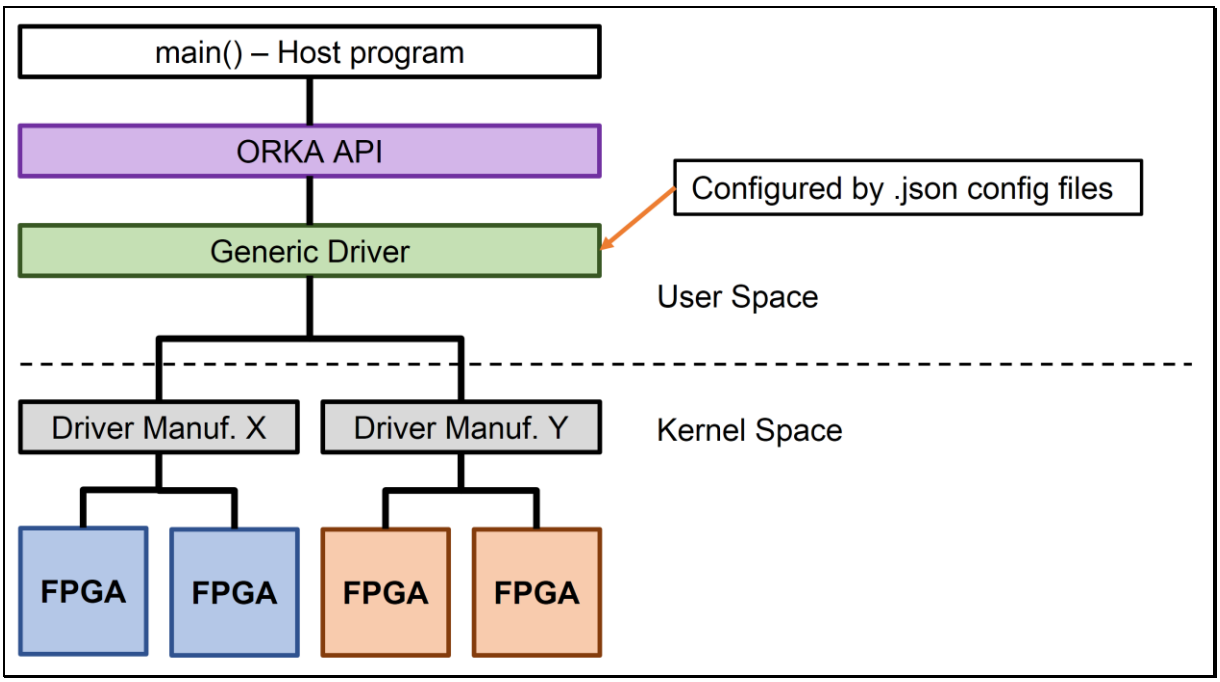
Inhalt

Einleitung 2
Problematik 2
Ursprünglicher Ansatz 2
Neue Idee 3
Lösung 3
Design-File 3
Designbeschreibungs-File 3
ORKAFPGAs.json 4
ORKAInterpreter.json 4
Beispiel 5
Konfigurationsfile 5
Software 7
Tools 9
ORKAVEC – Vektor-Container in C 9
ORKADB – InMemory-Datenbank 10
JSON 12

Einleitung

Nachdem jetzt etwa vier Wochen vergangen sind, hat sich einiges in der Softwarestruktur des GenericDriver Layers getan, was ich im Folgenden kurz ausführen möchte. Zu erwähnen ist noch, dass die Softwarestruktur zwar ziemlich weitgehend ausgearbeitet ist, jedoch noch nicht 100%ig fertig.

Unsere Architektur stellt sich im Augenblick wie folgt dar:



Es geht im Folgenden hauptsächlich um den GenericDriver in Grün.

Problematik

Die Problematik, die hier angegangen wird, ist mehrschichtig. Auf der einen Seite (nach oben im Schaubild) möchte man ein einheitliches Interface, auch wenn die unterschiedlichsten Treiber der nach Belieben verschiedenen Hersteller unterstützt werden sollen.

Auf der anderen Seite muss abstrahiert werden, dass alle Treiber ganz unterschiedliche Eigenschaften haben.

Weiterhin sind die Interfaces zu den FPGAs sehr unterschiedlich ausgeprägt, je nachdem welches Board sie beherbergt. Beispielsweise können mehrere FPGAs auf dem Board sein und das Board nur per Ethernet angebunden sein.

Die FPGAs selbst haben auch sehr unterschiedliche Eigenschaften oder Konfigurationen, die es unterstützen. Zum Beispiel müssen verschiedene Speicherbereiche zugänglich gemacht werden, genauso, wie auch Zusatzmodule, die mit dem FPGA angesprochen werden können (GPIO, …).

Ursprünglicher Ansatz

Ursprünglich sollte sich die Beschreibung der FPGAs in festen Datenstrukturen in dem GenericDriver befinden. Der Plan war, jedes Mal ein Release zu machen, wenn wir eine neues Board, andere FPGAs oder andere Hersteller unterstützen. Bei genauerer Betrachtung erschien das dann auf Dauer und im Großen betrachtet zu aufwändig.

Neue Idee

Die neue Idee ist eigentlich naheliegend, jedoch mit der Tücke im Detail. Man will Configfiles verwenden (Format zunächst egal).

Diese sollen dann die Konfiguration des aktuellen Designs beinhalten, wie z.B. die Speicherkonfiguration, das Interface zum Board (PCI, USB, o.ä.).

Das Problem ist, dass dieser Ansatz erstmal nicht generisch genug war, da es Probleme mit der herstellerübergreifenden Unterstützung gab. Aber zu jedem Problem gibt es auch Lösungen.

Lösung

Wir arbeiten mit verschiedenen Configfiles. Die Entscheidung fiel nachdem Teammitglieder .json bevorzugten auf JSON. Es hätte genauso gut aber auch XML sein können.

Das Ziel ist, möglichst herstellerübergreifend eine Beschreibung zu finden, die auch zukünftig möglichst wenig Änderungen erfordert und inhärent erweiterbar ist, ohne immer wieder Programmcode anfassen zu müssen. Das ist nur noch bei neu zu unterstützenden Herstellern in Ausnahmefällen erforderlich.

Die Dateien im Einzelnen:

1. Das Design-/die Design-file(s): <guid>.bit
2. Der Designbeschreibung: <guid>.json
3. Die FPGA Beschreibung: ORKAFPGAs.json
4. Die Interpretationsbeschreibung: ORKAInterpreter.json

Die .json Files können in beliebigen Verzeichnissen liegen, die beim **beliebig rekursiven Einlesen** dann gefunden werden!

Intern wird dann eine Datenbank mit mehreren Tabellen aufgebaut, die dann alle Informationen sammelt, damit sie dann nach oben hin zum ORKA API schnell abgerufen werden können.

Design-File

Das Designfile ist der Bitstream. Er ist im eigentlichen Sinne kein Konfigurationsfile. Das Besondere ist, dass wir uns für **GUID** (Globally Unique Identifier) entschieden haben, da diese eineindeutig sind. So können beliebig viele Designs in einem Verzeichnis untergebracht werden, ohne sich zu behindern.

Designbeschreibungs-File

Die Designbeschreibung enthält Angaben zum Board, das den FPGA beherbergt, wie z.B. den Herstellernamen oder einen Kommentar, der das Boarddesign beschreibt.

Es sollte **automatisiert** (nach einer Vorlage) mit dem Bitstream generiert werden.

Des Weiteren enthält es noch Angaben zu verwendbaren Treibern und Parameter dazu (z.B. die Instanz oder ein potentieller Port). Die PCIe Bars sind von entscheidender Bedeutung, wenn es um den Registerzugriff mittels PCIe geht.

Weiterhin müssen dann noch alle auf dem Board befindlichen FPGAs mit ihren zugehörigen Komponenten beschrieben werden. Insbesondere haben die Komponenten i.d.R. Register, für die man eine Basisadresse braucht. Unser ORKAIP, wie ich es hier mal nennen will, hat zusätzlich noch einen Offset im AXI Adressraum, den man noch setzen muss. Dieser orientiert sich an den Adressen, die der DRAM Controller dem angeschlossenen Speicher zuweist und das für jedes unterstützte Speicherinterface.

ORKAFPGAs.json

Dieses Konfigurationsfile ist händisch erstellt und sollte sukzessive erweitert werden, sobald neue FPGAs oder Hersteller unterstützt werden sollen.

Es unterscheidet zunächst zwischen den Herstellern und vergibt eindeutige IDs für diese.

Jeder FPGA benötigt eine eindeutige Bezeichnung und hat noch weitere Merkmale, die dann auf Abfrage aus der Datenbank oder internen Datenstrukturen einfach ausgelesen werden können.

ORKAInterpreter.json

Die Daten dieses Files stellen die Verbindung von beliebigen Bezeichnern und eineindeutigen und damit abfragbaren Begriffen her. Es enthält Boardkonfigurationen (Beschreibungen) und Übersetzungstabellen und ist der eigentliche Clou der Idee.

Dieses File ist händisch erstellt und wird bei neuen Boards, neuen oder geänderten Treibern ergänzt oder geändert. An der Software sollte dann nichts weiter geändert werden müssen!

Es werden Treiber der verschiedenen Art, wie z.B. JTAG, XDMA und QDMA angegeben, die einfach nur in dem Designbeschreibungsfile referenziert werden müssen.

Beispiel

Konfigurationsfile

Unten das Designbeschreibungs-File:

```
{
  "FileType": "boardsupportpackage",
  "Comment": "Design with access to 4GB Memory and little I/O",
  "BoardName": "Xilinx_VCU118_PCIE",
  "BlockDesignName": "axi_pcie_mig",
  "ManufacturerBoard": "Xilinx",

  "Drivers":
  [
    {
      "DriverName": "xdma",
      "Instance": 0,
      "Port": 0
    },
    {
      "DriverName": "qdma",
      "Instance": 0,
      "Port": 0
    }
  ]
```

Unten das Interpretationsbeschreibungsfile

```
{
  "FileType": "BoardSupportPackageConfig",
  "BoardName": "Xilinx VCU118 Board",
  "ManufacturersBoard":
  [
    {
      "------------------------": "------------------------------",
      "Name": "Xilinx",

      "Drivers":
      [
        {
          "jtag":
          {
            "control": "",
            "memcpyd2h": "",
            "memcpyh2d": "",
            "register": ""
          }
        },
        {
          "xdma":
          {
            "control": "/dev/xdma/card%interface%/control",
            "memcpyd2h": "/dev/xdma/card%interface%/c2h%dmachannel%",
            "memcpyh2d": "/dev/xdma/card%interface%/h2c%dmachannel%",
            "register": "/dev/xdma/card%interface%/user"
          }
        },
        {
          "qdma":
          {
            "control": "",
            "memcpyd2h": "",
            "memcpyh2d": "",
            "register": ""
          }
        },
```

Unten das Designbeschreibungs-File (Fortsetzung):

```
"FPGAs":
[
  {
    "Manufacturer": "Xilinx",
    "FullNameQualifier": "XCVU9P-LGA2104E",
    "Driver": 0,
    "Components":
    [
      {
        "name": "/xdma_0/M_AXI_LITE/SEG_axi_gpio_0_Reg",
        "offset": "0x00100000",
        "range": "0x00010000"
      },
      {
        "name": "/xdma_0/M_AXI_LITE/SEG_OR
        "offset": "0x01000000",
        "range": "0x00010000"
      },
      {
        "name": "/xdma_0/M_AXI/SEG_ddr4_0_
        "offset": "0x0000000100000000",
        "range": "0x0000000100000000"
      },
      {
        "name": "/ORKAProcessHW/Data_m_axi
        "offset": "0x0000000100000000",
        "range": "0x0000000100000000"
      }
    ]
  }
],
```

Der blaue Pfeil zeigt das Mapping an. Links die automatisch erzeugte Datei, unten unser Interpreter-File, das immer fix und unverändert bleibt.

```
"Components":
[
  {
    "type": "gpio",
    "address": "/xdma_?/M_AXI_LITE/SEG_axi_gpio_0_Reg",
    "subtype": "pushbutton",
    "access": "register"
  },
  {
    "type": "gpio",
    "address": "/xdma_?/M_AXI_LITE/SEG_axi_gpio_1_Reg",
    "subtype": "leds",
    "access": "register"
  },
  {
    "type": "memory",
    "address": "/xdma_?/M_AXI/SEG_ddr4_?_C?_DDR4_ADDRESS_BLOCK",
    "subtype": "ddr4",
    "access": "memcpy"
  },
  {
    "type": "orkaip",
    "address": "/xdma_?/M_AXI_LITE/SEG_ORKAProcessHW_Reg",
    "subtype": "aximaster",
    "access": "register"
  },
  {
    "type": "orkaip",
    "address": "/ORKAProcessHW/Data_m_axi_gmem/SEG_ddr4_?_C?_DDR4_ADDRESS_BLOCK",
    "subtype": "aximaster",
    "access": "memcpy"
  }

]
```

Wenn also der Vergleich positiv ist (das ‚?' steht für eine beliebige Integerzahl), dann bedeutet das, dass die Komponente ein „gpio" ist, Subtype „pushbutton" und der Zugriff darauf mittels „register" (also bei PCIe mit MMIO) erfolgt.

Type „memory" steht für Speicherblock, der per `ORKAGD_MemcpyH2D()` oder `ORKAGD_MemcpyD2H()` erreicht werden kann.

Im Falle, dass für einen Typ nicht „memory" steht, bedeutet das, dass es sich um normales IP mit Register (subtype „register") oder Speicherblock (subtype „memcpy") handelt, wobei auch beides gleichzeitig der Fall sein kann. Die jeweiligen Adressen dazu, kommen aus dem Designbeschreibungsfile und sind ja designabhängig. Oben im Beispiel sieht man, dass „orkaip" beides unterstützt, denn es gibt ein Muster mit „register" subtype und eines mit „memcpy", oben sogar mit Support für zwei DRam Bänke an zwei Memory Controllern.

Da die Strings unterschiedlich pro Hersteller sind, gibt es natürlich herstellerabhängige Sektionen, in denen das dann eingetragen werden kann.

Software

Das API hat sich nur in kleinen Bereichen gegenüber der ursprünglichen Implementation geändert. Intern ist ja viel passiert. Aber das zeigt eigentlich, dass der Ansatz schon prinzipiell richtig war.

Der Aufruf ist wie folgt:

```
ORKAGD_EC_t rc = ORKAGD_Init();    // expensive call
ORKAGD_ConfigTarget_t targetConfig =
{
    "d5ea0bc8-ffc7-474b-a5cb-a357c8bb3067.json",
};
void *target = ORKAGD_TargetBoardListOpen( &targetConfig ); // expensive call
```

Die zu filternde Zielkonfiguration kann man später praktisch noch nach Belieben erweitern …

Zurück kommt ein void-Pointer, der Intern eine Datenstruktur ist, die eine Liste verwaltet. Diese kann durchiteriert werden, z.B. mit einer while()-Schleife:

```
while (( ORKAGD_TargetBoardListRead( target )))
```

Das Target FPGA kann dann aus den vom Board zur Verfügung gestellten FPGAs ausgewählt werden.

```
size_t numFPGAs = ORKAGD_GetNumFPGAs( target );
uint64_t some_index = 0;
void *pTargetFPGA = ORKAGD_FPGAHandleCreate( target, some_index ); // expensive call
```

Man kann die Infrastruktur abfragen, die mit dem FPGA assoziiert ist (neu!):

```
uint64_t numComponents = ORKAGD_ComponentsGetNumOf( pTargetFPGA );
const ORKAGD_FPGAComponent_t *compEntry = ORKAGD_ComponentsGetEntry( pTargetFPGA, i );
```

Man kann dann alle Komponenten auslesen und die zugehörigen Parameter im Host Programm verwenden (Memory Blöcke, SystemManagement Einheit, Interrupt Controller und ORKA IPs).

Die folgende Datenstruktur ist “Human Readable“ und enthält im Grunde eine Kombination der Angaben aus Config File mit Interpreter File und das pro Komponente:

```
typedef struct
{
    char *ipName;       // name of IP from config
    uint64_t ipOffset;  // memory address (or register base address)
    uint64_t ipRange;   // size of memory block in bytes
    char *ipType;       // type name from config
    char *ipSubType;    // subtype name from config
    char *ipAccess;     // access type (register/memcpy)
    char *ipPath;       // pathname of ip within design
    char *ipBitstream;  // <GUID.json> filename of bitstream
 } ORKAGD_FPGAComponent_t;
```

Wir erinnern uns jetzt an das Configfile:

```
"name": "/xdma_0/M_AXI_LITE/SEG_ORKAProcessHW_Reg",
"offset": "0x01000000",
"range": "0x00010000"
```

und Interpreter-File (alles 1:1 drin bis auf den Namen der Adresse, da ich den unpassend fand, in der Datenstruktur heisst er jetzt ipPath, wir sollten das in dem Config-File entsprechend auch ändern ➔ @Matthias):

```
"type": "orkaip",
"address": "/xdma_?/M_AXI_LITE/SEG_ORKAProcessHW_Reg",
```

```
"subtype": "aximaster",
"access": "register"
```

Zurzeit haben wir im Interpreter File folgende Typen vorgesehen (und das ist ganz flexibel erweiterbar ohne Änderung am Code des GenericDrivers!!):

"memory": Speicherblöcke
"gpio": LEDs, Switches, Pushbuttons, etc.
"sysmgmt": Systemmanagement Einheit
"intc": Interrupt Controller
"orkaip": Von ORKA erstelltes IP (HLS)

Der folgende Aufruf initialisiert die Treiber und mapped die Register der Komponenten:

```
uint32_t rv = ORKAGD_TargetOpen( pTargetFPGA );
```

Dann kann man eigentlich nach Belieben schon mit der Infrastruktur (die bislang bereits auf das Board geladen werden sein muss) arbeiten:

```
// send input data to FPGA
ORKAGD_MemcpyH2D(
    pTargetFPGA,            // void *target,
    memoryFPGAStart,        // uint64_t destDevice,
    pInputDataMem,          // void *srcHost,
    inputDataSize );        // uint64_t byteSize );

// get back output from FPGA
ORKAGD_MemcpyD2H(
    pTargetFPGA,            // void *target,
    pOutputDataMem,         // void *destHost,
    memoryFPGAStart,        // uint64_t srcDevice,
    outputDataSize );       // uint64_t byteSize );

// start FPGA-Acceleration
ORKAGD_AcceleratorBlockStart(
    pTargetFPGA,            // void *target );
    orkaBaseMemoryWorkingAddresses[ 0 ],
    orkaIPHandle );

// start FPGA-Acceleration
ORKAGD_AcceleratorBlockWait(
    pTargetFPGA,            // void *target );
    orkaIPHandle );
```

Das *orkaIPHandle* ist die Rückgabe aus der Funktion *ORKAGD_ComponentsGetEntry()*, beim Auftreten des entsprechenden IP-Blocks (unseres ORKA IPs zum Beispiel, den wir am Typ „orkaip“ erkennen).

```
const ORKAGD_FPGAComponent_t * orkaIPHandle;
```

Am Ende räumen wir dann nur noch auf:

```
ORKAGD_TargetClose( pTargetFPGA );
ORKAGD_Deinit();
```

Das war‘s schon.

Tools

Zur Vereinfachung des Handlings von vielleicht später sehr vielen Daten habe ich Tools entwickelt, die ich im Folgenden erkläre:

ORKAVEC – Vektor-Container in C

Um einfacher eine Datenbank erstellen zu können, habe ich hier in Anlehnung an die STD-Lib in C++ eine C-Bibliothek von Funktionen zur Vektor-Container Behandlung neu entwickelt. Sie umfasst folgende Funktionen:

```
ORKAVEC_Vector_t *ORKAVEC_Create( uint64_t elementSize );
void ORKAVEC_Destroy( ORKAVEC_Vector_t *vector );
void *ORKAVEC_PushBack( ORKAVEC_Vector_t *vector, void *value );
uint64_t ORKAVEC_Size( ORKAVEC_Vector_t *vector );
void *ORKAVEC_GetAt( ORKAVEC_Vector_t *vector, uint64_t index );
ORKAVEC_Iter_t *ORKAVEC_IterCreate( ORKAVEC_Vector_t *vector );
void ORKAVEC_IterDestroy( ORKAVEC_Iter_t *iter );
void *ORKAVEC_IterBegin( ORKAVEC_Iter_t *vectorIterator );
void *ORKAVEC_IterNext( ORKAVEC_Iter_t *vectorIterator );
bool_t ORKAVEC_IterEnd( ORKAVEC_Iter_t *vectorIterator );
```

Man kann einen Vector erzeugen mit *ORKAVEC_Create()*. Hier übergibt man nur die Elementgröße und erhält ein Handle zurück, welches in den Folgefunktionen dann benutzt wird, um den richtigen Vektor zu identifizieren.

ORKAVEC_Destroy() zerstört ihn wieder,

ORKAVEC_PushBack() kopiert(!) das Element (memcpy) in das Array (den Vector-Container), wenn das zu klein ist, wird es dynamisch vergrößert um eine (intern) konfigurierbare Größe.

ORKAVEC_Size() gibt die Elementanzahl zurück,

ORKAVEC_GetAt() liefert einen Pointer auf das Element zurück.

ORKAVEC_IterCreate() erzeugt einen Iterator und gibt ein Handle darauf zurück mit dem man dann die nächsten Funktionen bedient:

ORKAVEC_IterDestroy() zerstört den Iterator wieder (Speicher Freigabe!).

ORKAVEC_IterBegin() gibt einen Pointer auf das erste Element des Vektors zurück,

ORKAVEC_IterNext() schaltet intern auf das nächste Element und gibt einen Pointer darauf zurück. Es führt zu Fehlern, wenn über das letzte Element hinaus diese Funktion aufgerufen wird, daher immer schön mit

ORKAVEC_IterEnd() abfragen, ob das Ende mit der Weiterschaltung erreicht wurde. Der Pointer ist dann ungültig.

Mit diesem Rüstzeug kann man dann die Datenbank in Angriff nehmen …

ORKADB – InMemory-Datenbank

Die „Datenbank“ ist jetzt keine „professionelle SQL-Datenbank“, jedoch erfüllt sie weitestgehend unsere Zwecke und wir können nach Belieben erweitern und optimieren.

```
ORKADB_DBHandle_t *ORKAGD_DBCreate( const char *databaseName );
ORKADB_DBHandle_t *ORKAGD_DBOpen( const char *databaseName );
void ORKADB_DBDestroy( ORKADB_DBHandle_t *databaseHandle );
ORKADB_TableHandle_t *ORKADB_TableCreate( ORKADB_DBHandle_t *databaseHandle, char
*tableName );
ORKADB_TableHandle_t *ORKADB_TableOpen( ORKADB_DBHandle_t *databaseHandle, char
*tableName );
void ORKADB_TableDestroy( ORKADB_TableHandle_t *tableHandle );
uint64_t ORKADB_TableNumEntries( ORKADB_TableHandle_t *tableHandle );
ORKADB_FieldHandle_t *ORKADB_FieldCreate( ORKADB_TableHandle_t *tableHandle, const
char *fieldName, ORKADB_FieldTypes_t fieldType, ORKADB_FieldOptions_t fieldOptions );
void ORKADB_FieldDestroy( ORKADB_FieldHandle_t *fieldHandle );
ORKADB_FieldHandle_t *ORKADB_FieldOpen( ORKADB_TableHandle_t *tableHandle, const char
*fieldName );
void ORKADB_FieldClose( ORKADB_FieldHandle_t *fieldHandle );
ORKADB_RecordHandle_t *ORKADB_RecordCreate( ORKADB_TableHandle_t *tableHandle );
void ORKADB_EntryFieldValueSetCopy( ORKADB_RecordHandle_t *entryHandle,
ORKADB_FieldHandle_t *fieldHandle, void *value );
void *ORKADB_RecordDataGet( ORKADB_RecordHandle_t *recordHandle );
void *ORKADB_RecordGetAt( ORKADB_TableHandle_t *tableHandle, uint64_t index );
void *ORKADB_RecordGetFieldByHandle( ORKADB_RecordHandle_t *record,
ORKADB_FieldHandle_t *field );
ORKAVEC_Vector_t *ORKADB_RecordListCreate( ORKADB_TableHandle_t *tableHandle,
ORKADB_FieldHandle_t *fieldHandle, void *searchData );
void ORKADB_RecordListDestroy( ORKAVEC_Vector_t *recordListCreate );
```

Ich glaube, dass die Namen eigentlich selbsterklärend sind, daher hier nur in Kürze:

Datenbank erzeugen, öffnen und zerstören:

ORKAGD_DBCreate(),

ORKAGD_DBOpen(),

ORKADB_DBDestroy()

Tabellen dieser Datenbank erzeugen, öffnen, zerstören und Anzahl Records der Tabelle:

ORKADB_TableCreate(),

ORKADB_TableOpen(),

ORKADB_TableDestroy(),

ORKADB_TableNumEntries()

Felder in der Tabelle erzeugen, öffnen, zerstören, schliessen:

```
ORKADB_FieldCreate( ORKADB_TableHandle_t *tableHandle, const char *fieldName,
ORKADB_FieldTypes_t fieldType, ORKADB_FieldOptions_t fieldOptions );
```

```
ORKADB_FieldDestroy(),
ORKADB_FieldOpen(),
ORKADB_FieldClose()
```

Erklärungsbedürftig scheint hier die zum Erzeugen eines Feldes nötigen Parameter:

Es werden verschiedene Typen unterstützt:

```
typedef enum
{
    ORKADB_FieldType_UnknownPointer = 0,
    ORKADB_FieldType_ValueI8,
    ORKADB_FieldType_ValueI16,
    ORKADB_FieldType_ValueI24,
    ORKADB_FieldType_ValueI32,
    ORKADB_FieldType_ValueI64,
    ORKADB_FieldType_ValueU8,
    ORKADB_FieldType_ValueU16,
    ORKADB_FieldType_ValueU24,
    ORKADB_FieldType_ValueU32,
    ORKADB_FieldType_ValueU64,
    ORKADB_FieldType_ValueF32,
    ORKADB_FieldType_ValueF64,
    ORKADB_FieldType_StringC,
    ORKADB_FieldType_Table
} ORKADB_FieldTypes_t;
```

Diese sind alle praktisch selbsterklärend bis auf UnknownPointer, StringC und Table:

UnknownPointer ist ein beliebiger Pointer, StringC ist einfach ein Zeiger auf ein char-Array, welches mit einem 0-Byte abgeschlossen ist und Table ist ein Tabellen-Handle dieser Datenbank.

Record-Handling:

ORKADB_RecordCreate(),

ORKADB_RecordDataGet(),

ORKADB_RecordGetAt(),

ORKADB_EntryFieldValueSetCopy(),

ORKADB_RecordGetFieldByHandle()

Sobald ein Record erzeugt ist für eine Tabelle, können keine Felder mehr hinzugefügt werden! Das wird verhindert. Diese Einschränkung kann man bei Bedarf herausprogrammieren, aber so war es jetzt einfacher.

Es wird immer mit Zeigern auf den Record gearbeitet. Der Record ist selbst eine Datenstruktur, die noch nicht die Daten enthält sondern nur einen Zeiger darauf.

Mit *ORKADB_EntryFieldValueSetCopy()* wird der Inhalt eines Datenblocks in ein Feld kopiert. Man übergibt immer einen void Zeiger auf die Daten, auch wenn es z.B. Pointer sind, also einen Pointer auf einen Pointer oder bei Integers einen Zeiger auf diesen.

Einen Zeiger zurück auf die Felddaten bekommt man mit der Funktion *ORKADB_RecordGetFieldByHandle()*.

Einen Vektor (siehe vorheriges Kapitel) aus allen Records einer Tabelle, die auf ein bestimmtes Suchmuster in einem Feld passen, bekommt man mit der Funktion *ORKADB_RecordGetFieldByHandle()*. Die Daten, die zum Vergleich herangezogen werden, müssen identisch sein.

Eine Ausnahme bildet der Typ StringC, hier kann der Vergleichsstring ‚?' enthalten, die dann an der entsprechenden Stelle im eigentlichen Feld durch beliebige positive Integerzahlen repräsentiert werden.

Wenn der Vergleichsstring also „abc?def" heisst, passt dieser auf

„abc0def", „abc1000def", abc164384763483146328473214937249374932 1def"

Aber nicht auf

„abc1.23def" oder „abc-1234def"

JSON

Hier wird eine freie Version eines einfachen JSON Parsers unter MIT Lizenz genommen. Es kann aber jeder andere sehr einfach auch eingesetzt werden.

```
<https://github.com/rafagafe/tiny-json>

  Licensed under the MIT License <http://opensource.org/licenses/MIT>.
  SPDX-License-Identifier: MIT
  Copyright (c) 2016-2018 Rafa Garcia <rafagarcia77@gmail.com>.
```

Horray ☺! Das wars.