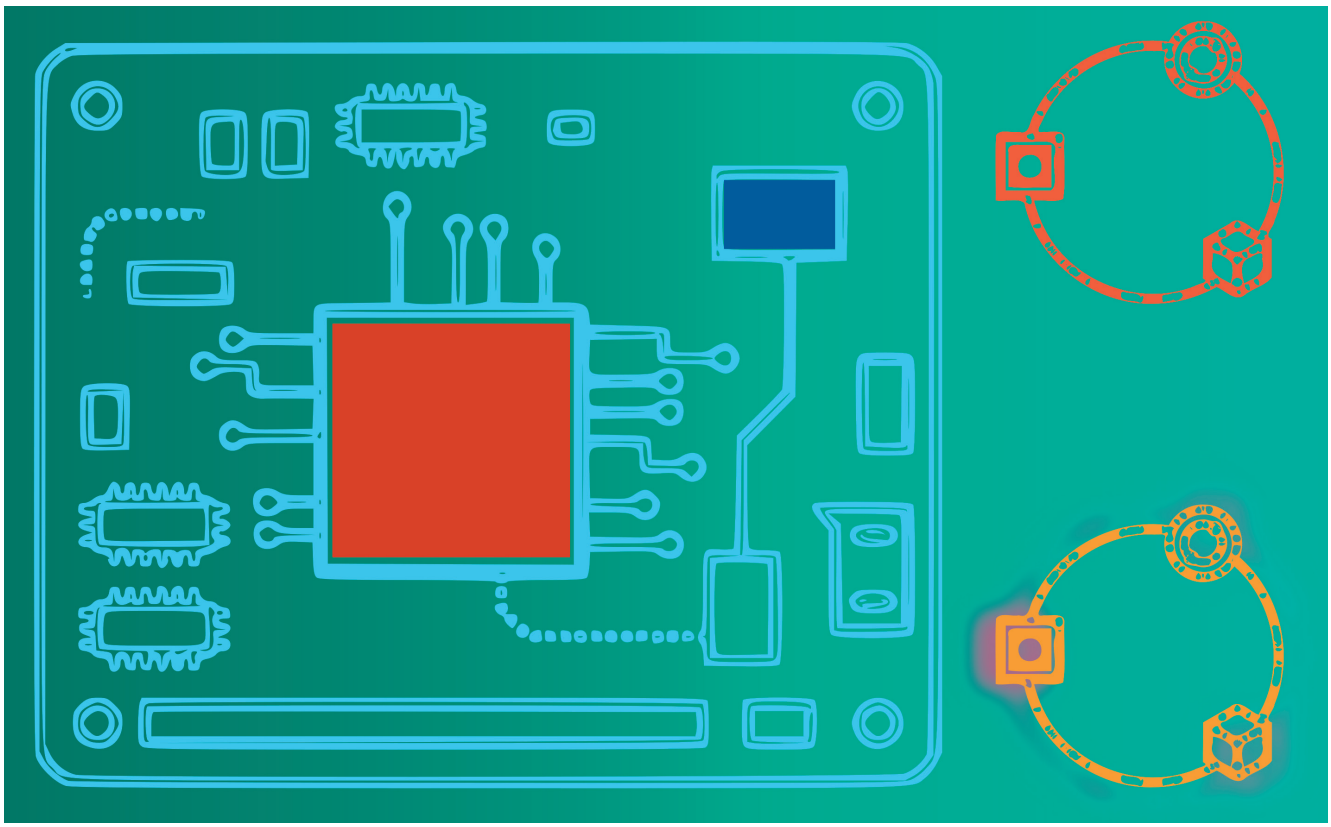


AZURE SPHERE, TEIL 2

Sphere spricht Hardware

Sicherheit und Komfort gelten als klassische Antipoden. Azure Sphere hat sich als bequem handhabbar erwiesen – als nächstes Problemfeld untersuchen wir die Echtzeitfähigkeit.

Microsoft sieht Azure Sphere – unter anderem – als klassisches Embedded-Betriebssystem, das im Markt quasi eine Ebene unter Windows 10 for IoT, Android Things und Co. angesiedelt ist. Derartige Elemente müssen im Allgemeinen mit einer Vielzahl von Peripheriegeräten kommunizieren, was sowohl GPIO als auch fortgeschrittene Hardwarebus-Systeme erfordert.

Angemerkt sei hierzu, dass Sphere zum Zeitpunkt der Drucklegung dieser Heftausgabe ausschließlich die klassischen GPIO-Pins und UART-Schnittstellen unterstützt. Die im MT3620 implementierten I2C- und SPI-Schnittstellen liegen mangels APIs brach.

Boardhersteller Seeed versucht, dieses Problem mit einem Erweiterungsboard auf UART-Basis zu kompensieren. Die in Bild 1 gezeigte Platine enthält eine Gruppe von Transceiver-Chips, die diverse Hardware-Busse nachbilden.

Ob sich die Umsetzung dieser Struktur in der Praxis lohnt, ist fraglich – es ist voraussichtlich nur eine Frage der Zeit, bis Microsoft die diversen Azure-Sphere-Schnittstellen nativ unterstützen wird.

GPIO im Blick

Das im vorangegangenen Artikel [1] realisierte Beispielprogramm beschränkte sich auf das langsame Blinken einer Leuchtdiode unter Timersteuerung. Zum Testen der realen GPIO-Performance ist diese Vorgehensweise denkbar ungeeignet – ein zweckmäßigerer, aber auch sehr roher Weg bestünde wohl darin, das Ein- und Ausschalten in einer Endlosschleife zu erledigen.

Im Interesse der Bequemlichkeit erzeugen wir jedoch ein neues Beispiel auf Basis der Vorlage *Blink Sample for MT3620 RDB*. Im Beispielcode zum Heft trägt das Projekt den Namen *Mt3620BlinkBenchmark*. Suchen Sie darin nach der in [1] besprochenen Methode *InitPeripheralsAndHandlers* und entfernen Sie den gesamten für die Einrichtung des Timers und des Eingangspins erforderlichen Code.

```
static int InitPeripheralsAndHandlers(void)
  {
  struct sigaction action;
  ...
```

```
  Log_Debug("Opening MT3620_RDB_LED1_RED.\n");
  gpioLedFd = GPIO_OpenAsOutput(MT3620_RDB_LED1_RED,
    GPIO_OutputMode_PushPull, GPIO_Value_High);
  if (gpioLedFd < 0) {
    ...
  }
  return 0;
}
```

Die Methode *main()* ist für das Ein- und Ausschalten des GPIO-Pins verantwortlich. Wir wollen das Problem durch eine Endlosschleife lösen, die jedoch eingehende Signale mitberücksichtigt:

```
int main(int argc, char *argv[])
{
  ...
  while (!terminationRequired) {
    if (WaitForEventAndCallHandler(epollFd) != 0) {
      terminationRequired = true;
    }
    GPIO_SetValue(gpioLedFd, GPIO_Value_High);
    GPIO_SetValue(gpioLedFd, GPIO_Value_Low);
    GPIO_SetValue(gpioLedFd, GPIO_Value_High);
    GPIO_SetValue(gpioLedFd, GPIO_Value_Low);
  }
  ClosePeripheralsAndHandlers();
  Log_Debug("Application exiting.\n");
  return 0;
}
```

Microsoft startet das Sphere-API mit vergleichsweise umfangreichen Fehlerberichterstattungsmethoden aus. Wer – wie der Autor – aus Nachlässigkeit vergisst, den *epoll*-Deskriptor anzulegen, sieht sich in Visual Studio mit folgenden Warnungen konfrontiert:

```
Blink application starting.
Opening MT3620_RDB_LED1_RED.
ERROR: Failed waiting on events: Bad file
  descriptor (9).
Closing file descriptors.
Application exiting.
```

Das Besondere an dieser Situation ist, dass die ausgegebene Fehlermeldung nicht im Quellcode von *main.c* vorkommt – sie stammt aus Microsofts Sphere-API. Nach der Beseitigung der Probleme schicken wir das Programm im vorliegenden Zustand auf den Prozessrechner. Leider leuchtet keine der vier Leuchtdioden auf. Zur Analyse der Ursachen bietet es sich an, den Ausgabepin durch eine Änderung der Arbeitsschleife immer auf *High* zu setzen:

```
  while (!terminationRequired) {
    if (WaitForEventAndCallHandler(epollFd) != 0)
    {
      terminationRequired = true;
    }
    GPIO_SetValue(gpioLedFd, GPIO_Value_High);
    //GPIO_SetValue(gpioLedFd, GPIO_Value_Low);
    //GPIO_SetValue(gpioLedFd, GPIO_Value_High);
    //GPIO_SetValue(gpioLedFd, GPIO_Value_Low);
  }
```

Leider bringt auch diese Programmversion keine nennenswerte Verbesserung. Die Ursache liegt in der Funktion *WaitForEventAndCallHandler*, die wartet, bis ein Event eingeht.

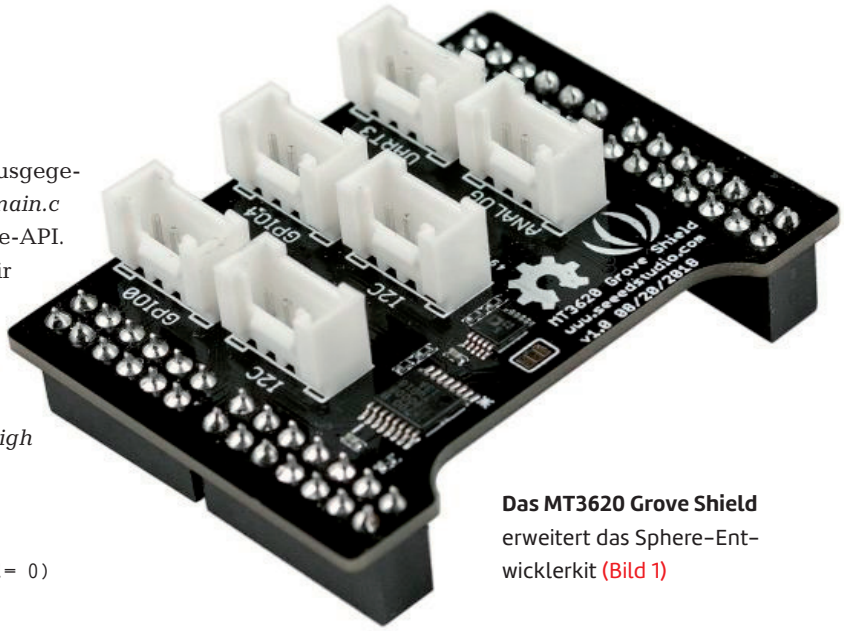
Da wir derzeit keinerlei Ereignisgeneratoren haben, retourniert auch die Funktion nie – das Resultat ist, dass der Code nicht abgearbeitet wird und das Board untätig wirkt.

Man könnte nun pragmatisch in dem Programm die Selektion zur Gänze auskommentieren, um als Payload nur noch das Ausgeben der charakteristischen Wellenform zu erhalten. Natürlich ist das in der Praxis nicht sehr zweckmäßig – die diversen vom Betriebssystem eingehenden Ereignisse sind für den Applikationscode in dieser Betriebsart verloren.

Für die folgenden Experimente wollen wir dieses Risiko jedoch auf uns nehmen. Wichtiger ist es, zu erfahren, an welcher Stelle wir das LED-Signal mit einem Oszilloskop abgreifen können. Hierzu müssen wir die Konstante *MT3620_RDB_LED1_RED* in Visual Studio rechts anklicken und der IDE den Befehl zur Anzeige der Definition geben. Visual Studio setzt uns daraufhin in der Datei *mt3620_rdb.h* ab, wo wir die Zuweisung an den GPIO-Pin Nummer acht sehen:

```
/// <summary>LED 1 Red channel is GPIO8.</summary>
#define MT3620_RDB_LED1_RED MT3620_GPIO8
```

Für uns ist die Situation insofern unerfreulich, als sich die beiden Header unseres Boards wie in Bild 2 und Bild 3 gezeigt präsentieren. Es ist offensichtlich, dass Seeed die mit Leuchtdioden und anderen Peripheriegeräten beschalteten Pins nicht ansprechbar macht. ▶



Das MT3620 Grove Shield erweitert das Sphere-Entwicklerkit (Bild 1)

Ein Weg zur Lösung dieses Problems besteht darin, in der Methode *InitPeripheralsAndHandlers* anstelle der Konstante *MT3620_RDB_LED1_RED* den Wert für den Pin Nummer fünf zu wählen. Er findet sich in bequem zugänglicher Position auf der Platine:

```
gpioLedFd = GPIO_OpenAsOutput(MT3620_GPIO5,
  GPIO_OutputMode_PushPull, GPIO_Value_High);
if (gpioLedFd < 0) {
```

Führen Sie das Programm danach abermals aus und wundern Sie sich nicht über das Dunkelbleiben der LED. Das liegt nicht daran, dass wir einen anderen Pin verwenden – in der Kommandozeilenausgabe von Visual Studio sehen Sie vielmehr eine vielsagende Warnung:

```
Opening MT3620_RDB_LED1_RED.
ERROR: Could not open LED GPIO: Permission
  denied (13).
Closing file descriptors.
```

GPIO in eingeschränkt

Der Zugriff auf externe Dienste löst bei sicherheitsbewussten Betriebssystemanbietern seit jeher Paranoia aus. So ist es dem Autor bis heute nicht verständlich, warum der Zugriff auf beliebige Sockets unter Firefox OS eine nur extrem schwierig zu erhaltende Sonderberechtigung voraussetzte. Im Fall von Azure Sphere ist die Situation insofern ähnlich, als der Entwickler in der Manifestdatei alle Pins anmelden muss, die seine Applikation während der Ausführung zu verwenden gedenkt.

Öffnen Sie dazu in Visual Studio die Datei *app_manifest.json*, um die von anderen Microsoft-Programmen bekannte JSON-Manifestdatei zu bearbeiten. Der für uns relevante Abschnitt trägt den Namen *Gpio* und das Anmelden der Pins braucht darin nicht in streng numerischer Reihenfolge zu erfolgen:

```
"Capabilities": {
  "AllowedConnections": [],
  "AllowedTcpServerPorts": [],
  "AllowedUdpServerPorts": [],
  "Gpio": [ 8, 9, 10, 12, 5 ],
  "Uart": [],
    ...
```

Nach dem Speichern der Datei ist das Programm abermals ausführungsbereit. Im Output-Fenster von Visual Studio erscheinen nun auch keine Fehlermeldungen mehr, weshalb wir zu professioneller Messtechnik greifen wollen.

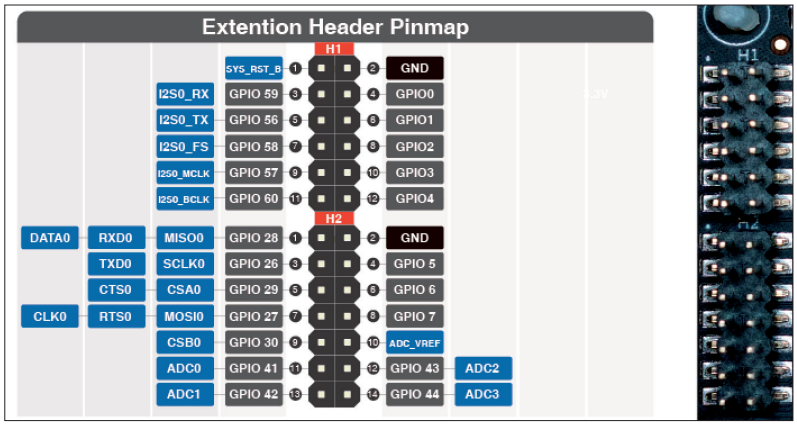
Von eminenter Bedeutung ist, die Verbindungen zwischen Sonde und Prozessrechner so kurz wie möglich zu halten. Wer auf die in jedem Labor in Massen vorhandenen Dupont-Verbindungen setzt, sieht sich mitunter mit dem in Bild 4 gezeigten Verhalten konfrontiert.

Wer den zunächst inkorrekten Anschluss nochmals neu und richtig vornimmt, erhält das in Bild 5 gezeigte Ergebnis.

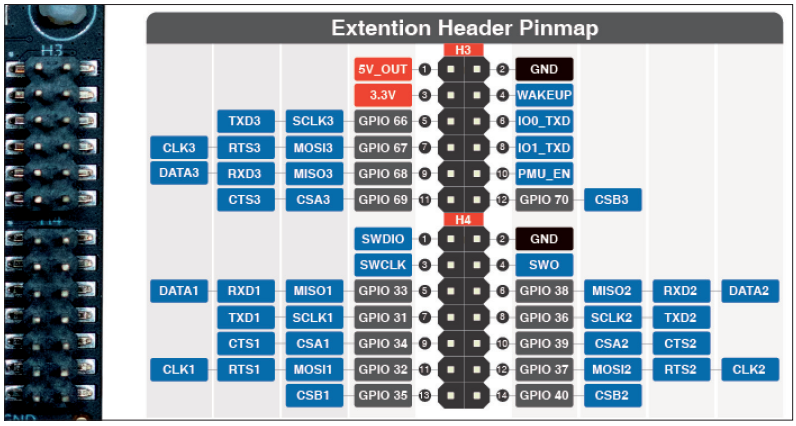
Die Unterschiede zwischen den beiden Wellentälern fallen übrigens nicht nennenswert ins Gewicht. Daraus folgt, dass der zur Bearbeitung der Schleife erforderliche Rechenaufwand sehr klein gegenüber dem ist, der für die Konfiguration der GPIO-Pins notwendig ist. Für die Qualifikation einer Ausgabewellenform ist ein Oszilloskop allerdings nur ein Werkzeug zweiter Klasse. Vernünftiger ist es, die Informationen – entweder per Jitter-Analyse oder aber durch einen dedizierten Modulationsdomänenanalysator – in ein Histogramm umzuwandeln. Die Ergebnisse dieser Vorgehensweise präsentieren sich dann wie in Bild 6 gezeigt.

Auf einem gewöhnlichen Prozessrechner würden wir auf einen der Ausgänge einen Interrupt festlegen, um uns von der Runtime beim Eintreffen des jeweiligen Signals informieren zu lassen. Ein digitaler Phosphor-Oszillograf ermöglicht dann das Messen der Verzögerung.

Microsoft macht uns die Arbeit an dieser Stelle insofern schwer, als man im unter [2] verlinkten Dokument darauf hinweist, dass ein Gutteil



Das Signal GPIO8 suchen Sie sowohl auf der linken … (Bild 2)



… als auch auf der rechten Erweiterungsbuchse vergebens (Bild 3)

der Hardware-Fähigkeiten des SOC nicht unterstützt wird. Für uns besonders relevant ist, dass die Interrupt-Funktionen der Eingänge brachliegen.

GPIO per Thread

Nach der Analyse des Verhaltens bei der Ausgabe von Wellenformen wollen wir feststellen, wie schnell der Prozessrechner auf eingehende Ereignisse reagiert.

Um die Situation so reaktiv wie möglich zu halten, bietet sich die Nutzung eines Threads an. Azure Sphere basiert, wie in [1] festgestellt, auf Linux und bietet deshalb eine mehr oder weniger vollständige Implementierung der POSIX-APIs an. Dieser wollen wir im Folgenden eine Chance geben.

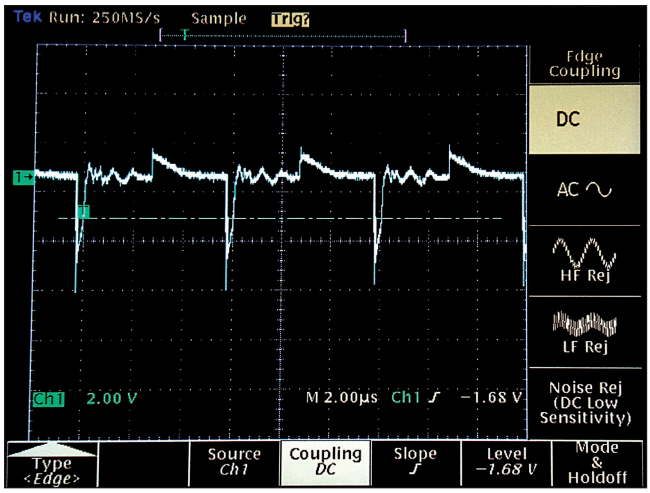
Die erste Aufgabe ist dabei das Feststellen der Thread-Sicherheit. Microsoft weist unter [3] explizit darauf hin, dass nur Teile der Azure-Sphere-API-Trainings sicher sind – welche Teile das genau sind, ist den jeweiligen Headern zu entnehmen.

Klicken Sie eine der GPIO-Funktionen in Visual Studio mit der rechten Maustaste an, um über die bekannte Deklarations-Anzeigefunktion in den enthaltenden Header zu gelangen. Im Fall der GPIO-Pins trägt die Datei den Namen *gpio.h* und in ihrem Kopf findet sich die folgende Passage:

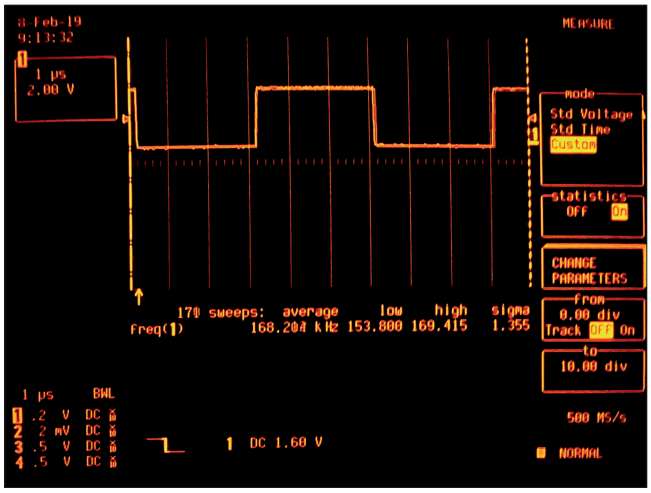
```
/// GPIO functions are thread-safe when accessing
/// different GPIOs concurrently. However, the caller
/// must ensure thread safety when accessing the same
/// GPIO.
```

Für uns bedeutet dies, dass wir es zwar nicht mit einem komplett sicheren API zu tun haben, in unserer spezifischen Situation aber keine Probleme auftreten sollten – das liegt daran, dass wir den Ausgabepin nur im Worker belasten.

Als ersten Akt zur Nutzung von POSIX-Threads müssen Sie den Header inkludieren. Dies mag für Unix-erfahrene Entwickler ungewohnt aussehen – Visual Studio enthält allerdings einen Cross-Compiler, weshalb das Einbinden wie gewohnt funktioniert:



Die seltsame Wellenform entsteht bei falschem Anschließen des Oszilloskops (Bild 4)



Auf dem korrekt angeschlossenen LeCroy-Oszilloskop sehen wir klarer (Bild 5)

```
#include <pthread.h>
```

Das POSIX-API erwartet die Anlieferung der Payload in Form einer *void*-Methode. Fürs Erste wollen wir eine Rechteckquelle ausgeben – da wir weiter oben festgestellt haben, dass die Schleife nicht nennenswert ins Gewicht fällt, können wir den Arbeiter folgendermaßen aufbauen:

```
void worker(void *ptr)
{
  while (1 == 1)
  {
    GPIO_SetValue(gpioLedFd, GPIO_Value_High);
    GPIO_SetValue(gpioLedFd, GPIO_Value_Low);
  }
}
```

Für seine Aktivierung ist dann – wie gewohnt – etwas Code in *main* erforderlich:

```
int main(int argc, char *argv[])
{
  Log_Debug("Blink application starting.\n");
  if (InitPeripheralsAndHandlers() != 0) {
    terminationRequired = true;
  }

  pthread_t myWorkerData;
  int threadNum;
  threadNum = pthread_create(&myWorkerData, NULL,
    worker, (void*)NULL);
```

Viele POSIX-Funktionen erwarten von ihrem Aufrufer einen Speicherbereich, der Zwischenergebnisse aufnimmt. Das ist auch bei der Funktion *pthread_create* der Fall, die Arbeitsstruktur trägt den Namen *pthread_t*.

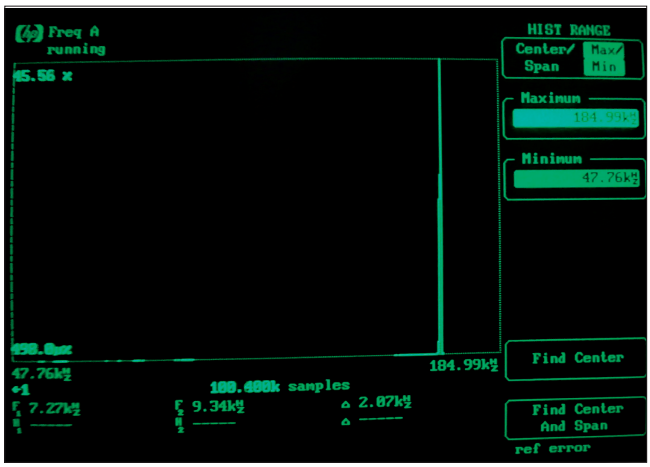
Während der Programmausführung zeigt sich am Modulationsdomänen-Analysator das in Bild 7 gezeigte und zuge- ▶

gebenermaßen etwas seltsame Erscheinungsbild. Aufmerksame Leser könnten vermuten, dass der Vergleich von eins gegen eins für die Performance-Verluste verantwortlich wäre. Dem ist allerdings nicht so, was sich durch die folgende Version der Schleife überprüfen lässt:

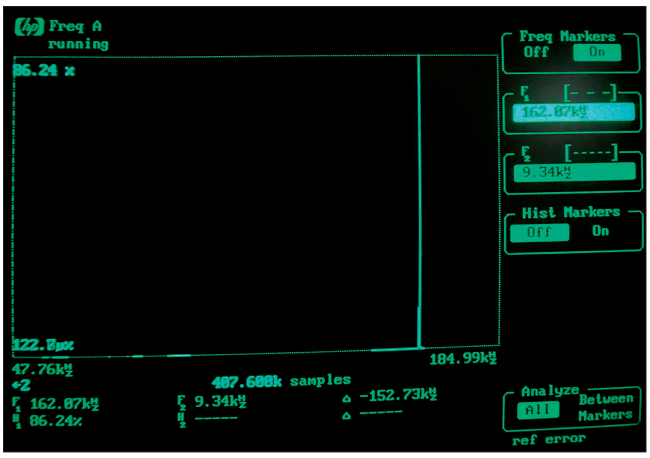
```
void worker(void *ptr)
{
  while (true)
  {
    GPIO_SetValue(gpioLedFd, GPIO_Value_High);
    GPIO_SetValue(gpioLedFd, GPIO_Value_Low);
  }
}
```

Führen Sie diese geänderte Version des Programms aus, so bekommen Sie ein noch seltsameres Ergebnis. Bild 8 zeigt, dass die Verwendung der einfacheren Bedingung zu einer noch langsameren Abarbeitung des Resultats führt.

Unsere letzte Aktion im Bereich der Pins besteht darin, den Sphere mit einer externen Quelle anzuregen und zu prüfen, wie lange unser Programm für das Entgegennehmen der Informationen braucht. Hierzu brauchen wir einen Eingang, der nach dem schon bekannten Schema entsteht – die Anmeldung im Manifest ist an dieser Stelle aus Platzgründen ausgelassen:



Die Informationen nach der Umwandlung ins Histogramm (Bild 6)



Die im Thread erzeugte Wellenform ist etwas langsamer als ihre reguläre Kollegin (Bild 7)

```
gpioButtonFd = GPIO_OpenAsInput(MT3620_GPIO42);
```

Das Einlesen des Pin-Wertes erfolgt über eine Funktion, die ebenfalls einen vom Entwickler bereitzustellenden Arbeitsbereich benötigt. Dieser lässt sich aber sofort an die Ausgabefunktion weiterreichen:

```
void worker(void *ptr)
{
  GPIO_Value_Type myVal;
  while (true)
  {
    GPIO_GetValue(gpioButtonFd, &myVal);
    GPIO_SetValue(gpioLedFd, myVal);
  }
}
```

Für die Anregung wollen wir auf einen Danaher-Funktionsgenerator vom Typ AWG2021 setzen. Seine Einstellungen zeigt Bild 9 – terminieren Sie die Verbindung mit einem 50-Ohm-Widerstand, um Überschießen zu verhindern.

Wenn Sie die Platine anschließen, erhalten Sie die Oszillogramm-Darstellungen in Bild 10 und 11. Das obere Signal ist die Antwort, die von der unten eingeblendeten Ausgabe des AWG angetrieben wird.

Angemerkt sei, dass diese Versuche aus Microsoft-Sicht nur sehr eingeschränkt gewünscht sind – wer harte Echtzeit-Performance benötigt, soll nach Microsofts Willen lieber auf den dedizierten Echtzeitprozessor umsteigen. Leider steht dieser der Allgemeinheit noch nicht zur Verfügung – interessierte Entwickler müssen an Microsoft eine Anfrage stellen.

Vom UART

Das in der Einleitung genannte „serielle" Board kommuniziert über einen UART mit dem Sphere. Da es sich dabei zum Redaktionsschluss um das wahrscheinlich leistungsfähigste Interface des Prozessrechners handelt, wollen wir als Nächstes einen kurzen Blick auf den zum Ansprechen erforderlichen Code werfen.

Als Basis dient uns dabei eine neue Instanz des Beispiels *Azure Sphere | UART Sample*, das Visual Studio Ihnen im Rahmen des Projektgenerators automatisch anzeigt.

Der erste Unterschied zum Basisprogramm ist, dass die Manifestdatei nun auch den UART *ISU0* als anzusprechendes Hardware-Element deklariert:

```
"Capabilities": {
  "AllowedConnections": [],
  "AllowedTcpServerPorts": [],
  "AllowedUdpServerPorts": [],
  "Gpio": [ 15, 16, 17, 12 ],
  "Uart": [ "ISU0" ],
```

Moderne UARTs halten auf Hardware-Ebene ein oder mehrere Register vor, die eingehende und ausgehende Informationen zwischenspeichern. Für Sie als Entwickler ist dies insofern relevant, als das Einschreiben beziehungsweise Lesen von Daten nicht unbedingt sofort zu Bewegung auf Hardware-Ebene führt. Jedenfalls ist für das Anmelden des *Uart*-Ereignishandlers eine weitere Eventdatenstruktur erforderlich. Sie entsteht als statischer Member, der einzige gesetzte Wert ist ein Verweis auf die eigentliche Methode:

```
static event_data_t uartEventData = {.eventHandler =
  &UartEventHandler};
```

Bei der Initialisierung bekommt der UART über ein Konfigurationsobjekt verschiedene Eigenschaften eingeschrieben. Besonders wichtig ist die Anmeldung der Baudrate:

```
UART_Config uartConfig;
UART_InitConfig(&uartConfig);
uartConfig.baudRate = 115200;
uartConfig.flowControl = UART_FlowControl_None;
```

Nach dem Einbuchen dieser Daten sind wir noch nicht am Ziel. Der UART wird zum Erhalt eines File Descriptors geöffnet, der Eventhandler wandert in die *EPOLL*-Schleife:

```
uartFd = UART_Open(MT3620_RDB_HEADER2_ISUO_UART,
  &uartConfig);
if (uartFd < 0) {
  Log_Debug("ERROR: Could not open UART: %s
    (%d).\n", strerror(errno), errno);
  return -1;
}
if (RegisterEventHandlerToEpoll(epollFd, uartFd,
    &uartEventData, EPOLLIN) != 0) {
  return -1;
}
```

Beim Eintreffen von Informationen müssen wir einen Puffer bereitstellen, der danach an die *read()*-Methode wandert. Sie nimmt einen Parameter mit dem maximalen im Puffer bereitstehenden Platz entgegen und retourniert die Anzahl der vom UART eingelesenen Bytes:

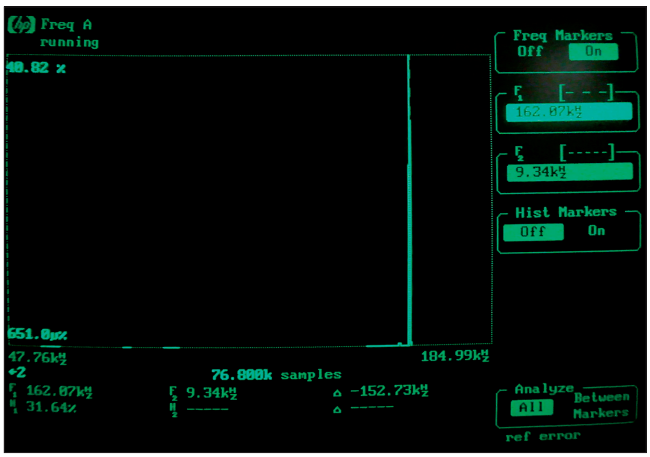
```
static void UartEventHandler(event_data_t *eventData) {
  const size_t receiveBufferSize = 256;
  uint8_t receiveBuffer[receiveBufferSize + 1];
  // allow extra byte for string termination
  ssize_t bytesRead;

  // Read UART message
  bytesRead = read(uartFd, receiveBuffer,
    receiveBufferSize);
  if (bytesRead < 0) {
    Log_Debug("ERROR: Could not read UART: %s (%d).\n",
      strerror(errno), errno);
    terminationRequired = true;
    return;
  }
```

Von einer Gegenstelle empfangene Informationen müssen nicht unbedingt nullterminiert sein. Microsofts Beispielcode handhabt dieses Risiko durch das manuelle Antackern einer Null – die restliche Verarbeitungsroutine drucken wir aus Platzgründen nicht ab:

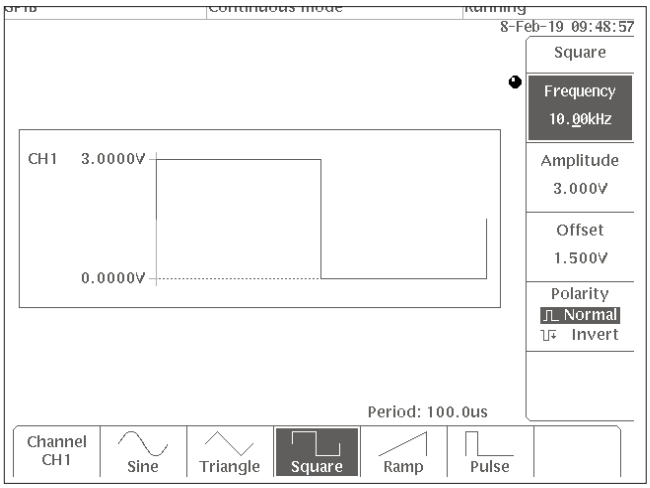
```
if (bytesRead > 0) {
  // Null terminate the buffer to make it a valid
  // string, and print it
  receiveBuffer[bytesRead] = 0;
```

Das Senden von Informationen ist insofern komplizierter, als unser Applikationscode nicht wirklich feststellen kann, wie viele Bytes noch im UART auf die Übertragung warten. Da die Write-Funktion Informationen über die gesendete Datenmenge liefert, spricht nichts dagegen, die Übertragung über eine Schleife zu forcieren:

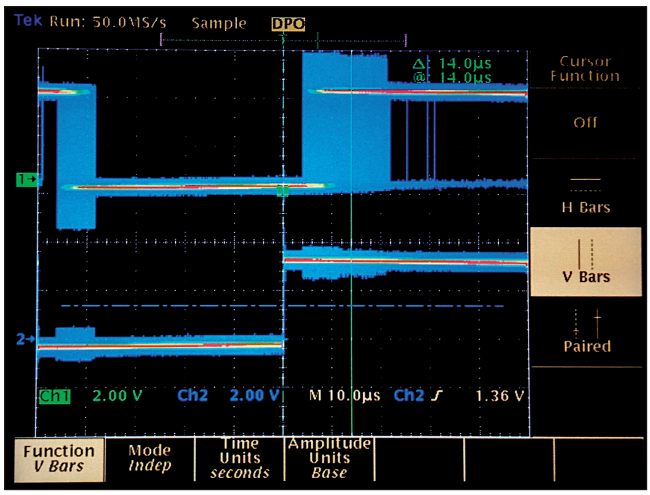
```
static void SendUartMessage(int uartFd, const char ▶
```



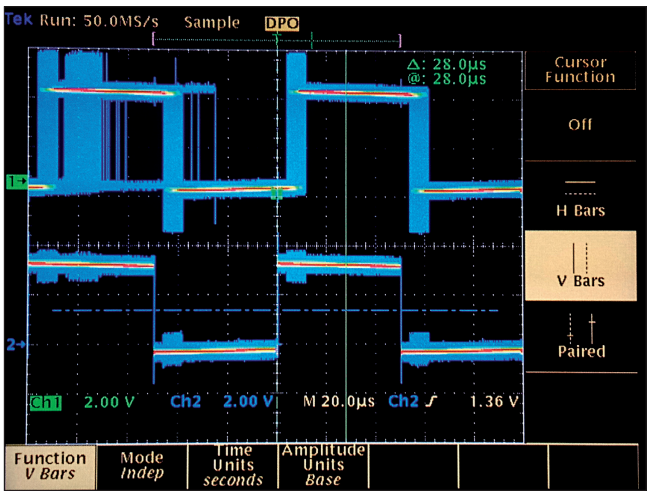
while(true) beschleunigt das Programm nicht (Bild 8)



Das anregende Signal darf nicht negativ werden (Bild 9)



Die normale Reaktionsverzögerung liegt im Bereich 14 Mikrosekunden … (Bild 10)



… leider wird sie manchmal (siehe blaue durchgehende Linie) auch so lang, dass ein Impuls verloren geht (Bild 11)

```
  *dataToSend)
  {
    size_t totalBytesSent = 0;
    size_t totalBytesToSend = strlen(dataToSend);
    int sendIterations = 0;
    while (totalBytesSent < totalBytesToSend) {
      sendIterations++;
      // Send as much of the remaining data as possible
      size_t bytesLeftToSend = totalBytesToSend
        - totalBytesSent;
      const char *remainingMessageToSend = dataToSend
        + totalBytesSent;
      ssize_t bytesSent = write(uartFd,
        remainingMessageToSend, bytesLeftToSend);
      if (bytesSent < 0) {
        Log_Debug("ERROR: Could not write to UART: %s
          (%d).\n", strerror(errno), errno);
        terminationRequired = true;
        return;
      }
      totalBytesSent += (size_t)bytesSent;
    }

    Log_Debug("Sent %zu bytes over UART in %d calls.\n",
      totalBytesSent, sendIterations);
}
```

Damit ist die UART-Routine hinreichend besprochen – die eigentliche Auswertung der angelieferten Daten erfolgt mit normalem C++-Code.

Fazit

Letztlich bleibt offen, was von der Echtzeitfähigkeit von Sphere zu halten ist. Wie schon Klaus Gims in seinem Klassiker zu Mikrobenchmarks [4] feststellt, gibt es kaum eine undankbarere Aufgabe als die Feststellung, ob ein System echtzeitfähig ist oder nicht. Offensichtlich ist, dass der MT3620 in der vorliegenden Form nicht zu 100 Prozent deterministisch reagiert – sicherheitskritische Systeme wie eine Motorsteuerung mit Propellersynchronisation würde der Autor mit ihm deshalb nicht realisieren. Andererseits ist die Leistung aber auch nicht so schlecht, dass man damit nur Trivialsteuerungen realisieren kann. Zudem ist im Moment alles andere als klar, ob Microsoft dem MT3620 nicht noch mit einem Betriebssystemupdate auf die Sprünge hilft. In der Theorie könnte man Code auf Kernelebene ausführen – dass dies der Sicherheit nicht zuträglich ist, ist allerdings logisch. Kurz: Es bleibt spannend, wie Microsoft das Spannungsfeld zwischen Performance und Sicherheit handhaben wird – bisher fährt man hier in Redmond einen nicht unbedingt schlechten Kurs.

Auch wenn es über die Hardware-Interfaces des Azure Sphere noch einiges zu berichten gäbe, wollen wir unsere diesbezüglichen Experimente an dieser Stelle beenden. In einem weiteren Artikel werden wir uns stattdessen der Interaktion zwischen Azure Sphere und den Azure-Clouddiensten zuwenden. ■

[1] Tam Hanna, Das Microsoft-Linux, dotnetpro 4/2019, S. 62 ff, www.dotnetpro.de/A1904Sphere
[2] MT3620 Support Status, www.dotnetpro.de/SL1905Sphere1
[3] Asynchronous events and concurrency, www.dotnetpro.de/SL1905Sphere2
[4] Klaus Gims, Mikrobenchmarks, www.dotnetpro.de/SL1905Sphere3

Tam Hanna
entwickelt Programme für verschiedene Plattformen, betreibt Online-Newsdienste zum Thema und steht für Fragen, Trainings und Vorträge gern zur Verfügung. Sie erreichen ihn unter der E-Mail-Adresse **tamhan@tamoggemon.com**.

dnpCode A1905Sphere