In der Vorlesung haben Sie eine UART als Beispiel für ein I/O-Gerät kennengelernt. Zur Erinnerung:

- Die UART hat 8 Register mit jeweils 8 Bit.
- Am Datenausgang der UART (verbunden mit dem Datenbus), werden die 8-Bit Worte mit 24 Nullen zu 32-Bit Worten aufgefüllt.
- Wenn eine Adresse $(a_{31}, a_{30}, ..., a_0)$ mit 01 beginnt, dann wird die UART angesprochen (Memory Map).
- Die Register werden mit den drei niederwertigsten Bits (a_2, a_1, a_0) vom Adressbus adressiert.

Wir nehmen an, dass die acht Register R0 - R7 der UART aufsteigend adressiert sind, d.h. R0 wird adressiert mit $010^{27}000$, R1 mit $010^{27}001$, usw..

Wir gehen jetzt von folgendem **vereinfachten Kommunikationsprotokoll** aus:

Wir betrachten die Register R0, R1 und R2 näher. Hierbei sei (aus CPU/ReTI-Sicht) R0 zum *Senden* von Daten an das Peripheriegerät (verbunden über UART), R1 zum *Empfangen* und R2 zum Lesen bzw. Schreiben von *Statusmeldungen*. In R2 hat jedes der acht Bit eine Statusfunktion. Der Einfachheit halber nehmen wir an, dass nur die beiden niederwertigsten Bits für uns relevant sind. Das niederwertigste Bit b0 von R2 wird wie folgt verwendet:

- b0 = 0 wenn das Senderegister (R0) von der CPU vollgeschrieben wurde. Die UART akzeptiert keine weiteren Daten mehr bis die aktuellen versendet wurden.
- b0 = 1 wenn das Senderegister von der CPU befüllt werden darf.

Das darauffolgende Bit b1 von R2 wird folgendermaßen verwendet:

- b1 = 0 wenn keine Daten im Empfangsregister (R1) sind bzw. diese schon von der CPU gelesen wurden.
- b1 = 1 wenn alle Daten von der Peripherie kommend im Empfangsregister (R1) bereit stehen.

Bitte führen Sie Ihren Code für die unten aufgeführten Arbeitsanweisungen mit dem RETI-Interpreter aus!

Folgendes Programm `program.reti`

```
# input: 42 3
INT 3 # just a breakpoint, use c in debug mode (-d) to directly jump here
INT 0
MOVE ACC IN1
INT 0
MULT ACC IN1
INT 1
```

`JUMP 0`

verwendet Ihre Interrupt-Service-Routinen. Benutzen Sie dieses, um Ihre Interrupt-Service-Routinen im RETI-Interpreter zu testen. Nutzen Sie hierzu folgenden Befehl:

```
$ reti_interpreter -i interrupt_service_routines.reti -d -m program.reti
```

Mithilfe der Kommandozeilenoption `-i` ist der RETI-Interpreter in der Lage die RETI-Befehle für **Interrupt-Service-Routinen** aus einer Datei `interrupt_service_routines.reti` herauszulesen und an den Anfang des simulierten SRAM, vor das geladene Programm aus `program.reti` zu schreiben. Mithilfe von `INT i` kann wie in der Vorlesung erklärt an den Anfang jeder dieser Interrupt-Service-Routinen `i` gesprungen werden. Mittels `RTI` kann am Ende einer Interrupt-Service-Routine wieder an die nächste Stelle im ursprünglichen Programm zurückgesprungen werden, an der dieses mittels `INT i` unterbrochen wurde.

Mittels der Direktive `IVTE i` können **Einträge einer Interrupt Vector Tabelle** mit der korrekten Startadresse einer Interrupt-Sevice-Routine erstellt werden. Die SRAM Konstante wird automatisch auf `i` draufaddiert.

Wir implementieren in dieser Aufgabe zwei Interrupt-Service-Routinen (ISR). Benutzen Sie für Ihre Interrupt-Service-Routinen in `interrupt_service_routines.reti` folgendes Grundgerüst:

```
0000    IVTE 2 # INT 0: Get user input over UART and save it in ACC
0001    IVTE <i> # INT 1: Print an integer stored in ACC over UART
0002    SUBI SP 3 # start of ISR 0 at relative address 2 + 2^31
0003    STOREIN SP DS 1
0004    STOREIN SP IN1 2
....    STOREIN SP IN2 3
i-03    # your reti code goes here
i-03    LOADIN SP DS 1
i-02    LOADIN SP IN1 2
i-01    LOADIN SP IN2 3
i+00    ADDI SP 3
i+01    RTI
i+02    SUBI SP 4 # start of ISR 1 at address <i> + 2^31
i+03    STOREIN SP DS 1
i+04    STOREIN SP IN1 2
i+05    STOREIN SP IN2 3
i+06    STOREIN SP ACC 4
i+07    LOADI DS 1048576
i+08    MULTI DS 1024
i+09    MOVE ACC IN2
i+10    LOADI ACC 4
i+11    STORE ACC 0
i+12    LOAD ACC 2
i+13    ANDI ACC -2
i+14    STORE ACC 2
i+15    # your reti code goes here
i+16    LOADIN SP DS 1
i+17    LOADIN SP IN1 2
i+18    LOADIN SP IN2 3
i+19    LOADIN SP ACC 4
i+20    ADDI SP 4
i+21    RTI
i+22
i+23
i+24
```

Die 4 8-Bit-Zahlen zu Übertratung eines 32-Bit-Ganzzahl werden im vom RETI-Interpreter simulierten Eingabegerät im **Big-Endian Format** übertragen, was für ihre Implementierung der **Interrupt-Service-Routine 0** (`ISR 0`) relevant ist. Nachdem Sie durch setzen des Bit b1 im **Statusregister** auf 0 dem simulierten Eingabegerät mitgeteilt haben für den Empfang von Eingaben bereit zu sein, wird der User nach einer Eingabe gefragt (falls nicht die Kommandozeilenoption `-m` aktiviert wurde). Nach einer Wartezeit kommen die Daten im **Empfangsregister** R1 an und b1 im Statusregister wird wieder auf 1 gesetzt.

Die **Interrupt-Service-Routine 1** (`ISR 1`) muss zu Anfang zur Einhaltung eines Protokolls die Zahl 4 versenden, um dem im RETI-Interpreter simulierten Anzeigegerät auf der anderen Seite der UART-Kommunikation mitzuteilen, dass als nächstes eine Ganzzahl übertragen wird. Dies wird Ihnen im vorgegebenen Grundgerüst bereits abgenommen. Das Anzeigegerät wird vom RETI-Interpreter simuliert, indem dieses nach Erhalt der von 4 8-Bit-Zahlen im **Big-Endian Format** die zusammengesetzte 32-Bit-Zahl in ihrem Terminal über `stdout` ausgibt. Nachdem Sie in das **Senderegister** R0 die gewünschten Daten geschrieben haben und im **Statusregister** R2 das Bit b0 auf 0 gesetzt haben, wird vom RETI-Interpreter eine Wartezeit für die Übertragung der Daten simuliert und erst nach dieser Wartezeit wird b0 wieder auf 1 gesetzt.

Tipp: Sie können dieser Wartezeit mittels der Kommandozeilenoption `-w 0` auf 0 setzen, um beim Debuggen nicht unnötig warten zu müssen. Ihr Programm muss allerdings auch mit einer beliebig langen Wartezeit umgehen können.

Tipp: Um beim Debuggen nicht immer selbst einen Input eingeben zu müssen können sie mittels der Kommandozeilenoption `-m` Leerzeichenseparierte Inputs aus dem Kommentar `# input: -42 3` am Anfang des Programms `program.reti` rauslesen.

a) Schreiben Sie eine ISR, die eine 32-Bit Ganzzahl aus der UART in 8-Bit Packete aufgeteilt empfängt und diese dann in an die passenden Stellen im `ACC` Register schreibt. Setzen sie dazu zunächst b1 im Statusregister auf 0, um dem vom RETI-Interpreter simulierten Eingabegerät zu signalisieren, dass sie für den Empfang von Eingaben bereit sind. Schreiben sie danach ReTI-Code, der in einer Schleife läuft und ständig überprüft ob b1 im Statusregister R2 auf 1 gesetzt wurde, d.h. ob der CPU über die UART Daten in R1 bereitgestellt wurden. Wenn Daten (8-Bit) bereitgestellt wurden, soll ihr Programm diese an die passende Stelle im `ACC` Register laden und b1 wieder auf 0 setzen (sodass die UART wieder neue Daten in R1 schreiben darf). Sie müssen weitere ähnliche Routinen für die verbleibenden 3 8-Bit Packete implementieren.

   Beachten Sie, dass Sie vor Zugriffen auf die UART im Datensegmentregister `DS` das entsprechende Präfix 01 benötigen. Sie müssen also die 32-Bit Konstante 010^{30} mit der Methode aus Aufgabe 1) in das `DS` Register bringen.

   Anmerkung: Interrupt-Service-Routinen müssen die ursprünglichen Registerinhalte auf dem Stack absichern. Nach Beendigung und vor Aufruf von `RTI` müssen diese dann wiederhergestellt werden.

b) Schreiben Sie eine zweite ISR, die eine im `ACC` gespeicherte Ganzzahl über die UART versendet. Überprüfen Sie dazu vor jedem Versenden, ob die UART bereit ist (b0 = 1) und das Senderegister befüllt werden darf. Dann befüllen Sie jeweils das Senderegister R0 mit 8-Bit Packeten des Wortes im `ACC` Register und denken jedes mal daran b0 des Statusregister danach auf

0 zu setzen. Sie müssen weitere ähnliche Routinen für die verbleibenden 3 8-Bit Packete implementieren.

`interrupt_service_routines.reti`

```
IVTE 2 # INT 0: Get user input over UART and save it in ACC
IVTE 53 # INT 1: Print an integer stored in ACC over UART
SUBI SP 2 # start of ISR 0
STOREIN SP DS 1
STOREIN SP IN1 2
LOADI DS 1048576
MULTI DS 1024
LOAD ACC 2
ANDI ACC -3
STORE ACC 2
LOAD ACC 2
ANDI ACC 2
JUMP== -2
LOAD ACC 1
MOVE ACC IN1
ANDI ACC 128
JUMP== 4
LOADI ACC -1
OPLUSI ACC 255
OR IN1 ACC
MULTI IN1 1048576
MULTI IN1 16
LOAD ACC 2
ANDI ACC -3
STORE ACC 2
LOAD ACC 2
ANDI ACC 2
JUMP== -2
LOAD ACC 1
MULTI ACC 65536
OR IN1 ACC
LOAD ACC 2
ANDI ACC -3
STORE ACC 2
LOAD ACC 2
ANDI ACC 2
JUMP== -2
LOAD ACC 1
MULTI ACC 256
OR IN1 ACC
LOAD ACC 2
ANDI ACC -3
```

```
STORE ACC 2
LOAD ACC 2
ANDI ACC 2
JUMP== -2
LOAD ACC 1
OR IN1 ACC
MOVE IN1 ACC
LOADIN SP DS 1
LOADIN SP IN1 2
ADDI SP 2
RTI
SUBI SP 4 # start of ISR 1
STOREIN SP DS 1
STOREIN SP IN1 2
STOREIN SP IN2 3
STOREIN SP ACC 4
LOADI DS 1048576
MULTI DS 1024
MOVE ACC IN2
LOADI ACC 4
STORE ACC 0
LOAD ACC 2
ANDI ACC -2
STORE ACC 2
LOAD ACC 2
ANDI ACC 1
JUMP== -2
MOVE IN2 IN1
LOADI ACC 2097151
MULTI ACC 1024
ORI ACC 1023
AND IN1 ACC
DIVI IN1 256
DIVI IN1 65536
MOVE IN2 ACC
JUMP>= 2
ORI IN1 128
STORE IN1 0
LOAD ACC 2
ANDI ACC -2
STORE ACC 2
LOAD ACC 2
ANDI ACC 1
JUMP== -2
MOVE IN2 IN1
LOADI ACC 2097151
```

```
MULTI ACC 1024
ORI ACC 1023
AND IN1 ACC
DIVI IN1 65536
STORE IN1 0
LOAD ACC 2
ANDI ACC -2
STORE ACC 2
LOAD ACC 2
ANDI ACC 1
JUMP== -2
MOVE IN2 IN1
LOADI ACC 2097151
MULTI ACC 1024
ORI ACC 1023
AND IN1 ACC
DIVI IN1 256
STORE IN1 0
LOAD ACC 2
ANDI ACC -2
STORE ACC 2
LOAD ACC 2
ANDI ACC 1
JUMP== -2
STORE IN2 0
LOAD ACC 2
ANDI ACC -2
STORE ACC 2
LOAD ACC 2
ANDI ACC 1
JUMP== -2
LOADIN SP DS 1
LOADIN SP IN1 2
LOADIN SP IN2 3
LOADIN SP ACC 4
ADDI SP 4
RTI
```