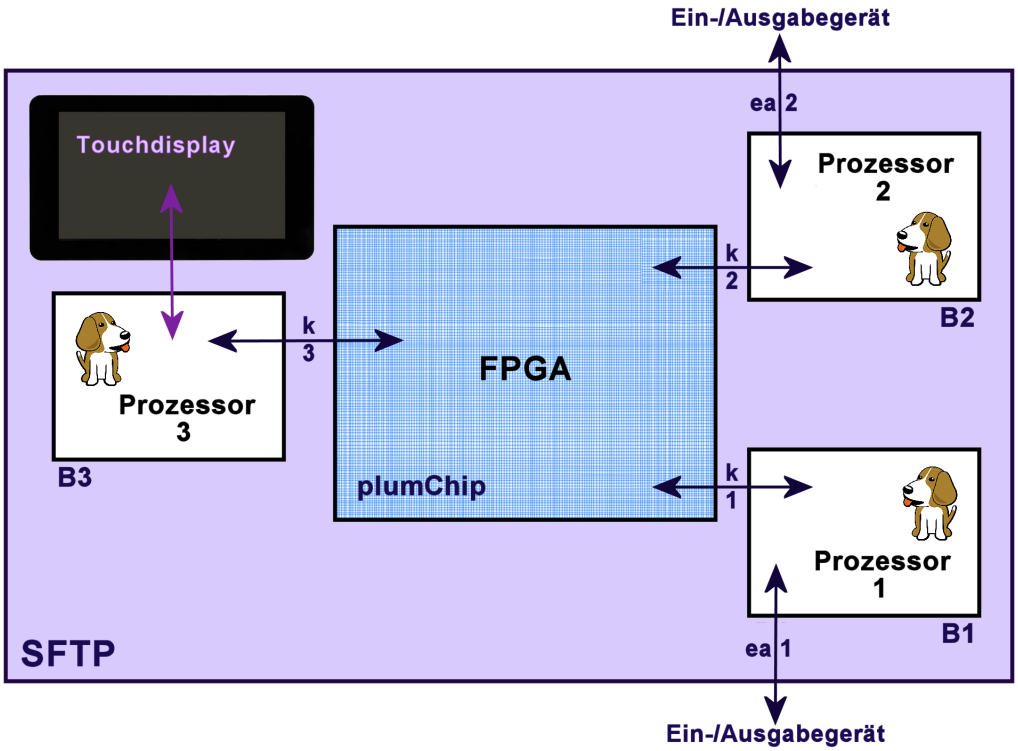
plumChip

Krypto-API, Version 1.8.1



Programmierschnittstelle

Beschreibung mit Beispielen

imbit.net

Stand: 2. Dezember 2019

Inhaltsverzeichnis

1 Allgemeine Hinweise **1**
- 1.1 Bereitschafts-, Erfolgs- und Misserfolgsanzeige . . . 2
- 1.2 Einrichten der seriellen Schnittstellen . . . 2
- 1.3 Senden und Empfangen über die seriellen Schnittstellen . . . 3
- 1.4 Zurücksetzen des `plumChips` . . . 4
- 1.5 Kontrollanzeigen . . . 4

2 Zugriff auf die Matrix des `plumChips` **4**
- 2.1 Löschen der Matrix . . . 4
- 2.2 Zufälliges Befüllen der Matrixspalten zur Hälfte mit Einsen . . . 5
- 2.3 Zufälliges Befüllen der Matrixzeilen zur Hälfte mit Einsen . . . 5
- 2.4 Weitere Matrixbefehle . . . 6

3 Zugriff auf die Register des `plumChips` **6**
- 3.1 Zufälliges Befüllen des Frageregisters zur Hälfte mit Einsen . . . 6
- 3.2 Eingabe ins Lernantwortregister . . . 7
- 3.3 Ausgabe des Lernantwortregisters . . . 7
- 3.4 Frage- und Lernantwortregister mit XOR verknüpfen . . . 7
- 3.5 Schlüssel erzeugen . . . 8
- 3.6 Salz holen . . . 9
- 3.7 Weitere Registerbefehle . . . 9

4 Verwaltungsbefehle **9**
- 4.1 Kennung des `plumChips` einlesen . . . 9
- 4.2 Sekundensignal ein- und ausschalten . . . 10
- 4.3 Zufallsgenerator initialisieren . . . 10
- 4.4 Inhalt des Zufallsgenerators anzeigen . . . 11

5 Kommunikationsbefehle **12**
- 5.1 Nachrichtenlänge festlegen . . . 12
- 5.2 Nachrichten an die Prozessoren senden . . . 12
  - 5.2.1 Prozessor 3 an Prozessor 1 oder 2 . . . 12
  - 5.2.2 Prozessor 1 an Prozessor 2 oder 3 . . . 13
  - 5.2.3 Prozessor 2 an Prozessor 1 oder 3 . . . 13
- 5.3 Ein- und Ausgaben der Prozessoren 1 und 2 . . . 14

6 Tabellarische Befehlsübersicht **17**
- 6.1 Matrixbefehle . . . 17
- 6.2 Registerbefehle . . . 18
  - 6.2.1 Frageregister Q . . . 18
  - 6.2.2 Lernantwortregister L . . . 18
- 6.3 Schlüsselgenerierung . . . 19
- 6.4 Verwaltungsbefehle . . . 19
- 6.5 Kommunikationsbefehle . . . 20

7 Literaturnachweis **20**

Dieses Projekt wird mit Mitteln des Europäischen Fonds für regionale Entwicklung gefördert.

EUROPÄISCHE UNION
Europäischer Fonds für regionale Entwicklung

EUROPA FÜR NIEDERSACHSEN
EFRE Förderung für die niedersächsischen Regionen

`plumChip` v1.8.1

Beschreibung der Programmierschnittstelle des Kryptobausteins („Krypto-API“)

Der den kryptografischen Zwecken dienende, zentrale Baustein der `aPlum`-Verschlüsselungshardware[1] befindet sich derzeit auf einer Entwicklungsplatine[2] in Form eines FPGA[3] und wird von drei Prozessoren[4] bedient. In der vorliegenden Version v1.8.1 sind die Prozessoren über UART-Schnittstellen[5] mit dem Kryptobaustein verbunden. Im Folgenden wird der Kryptobaustein als `plumChip` bezeichnet.

Vollen Zugriff auf den `plumChip` hat nur der Prozessor 3, der als `Beagle 3` bezeichnet wird.[6] Diese Beschreibung bezieht sich ganz überwiegend auf die Programmierung nur genau dieses Prozessors. Die anderen beiden Prozessoren `Beagle 1` und `Beagle 2` dienen der Abschirmung des `Beagle 3` von der Außenwelt, indem sie die notwendigen Ein- und Ausgaben von Klar- und Geheimdaten abwickeln. Die drei Prozessoren sind nicht direkt miteinander verbunden, sondern tauschen die nötigen Informationen über die Kommunikationsbefehle des `plumChip`s untereinander aus (siehe Kapitel 5.2 und Tabelle 5).

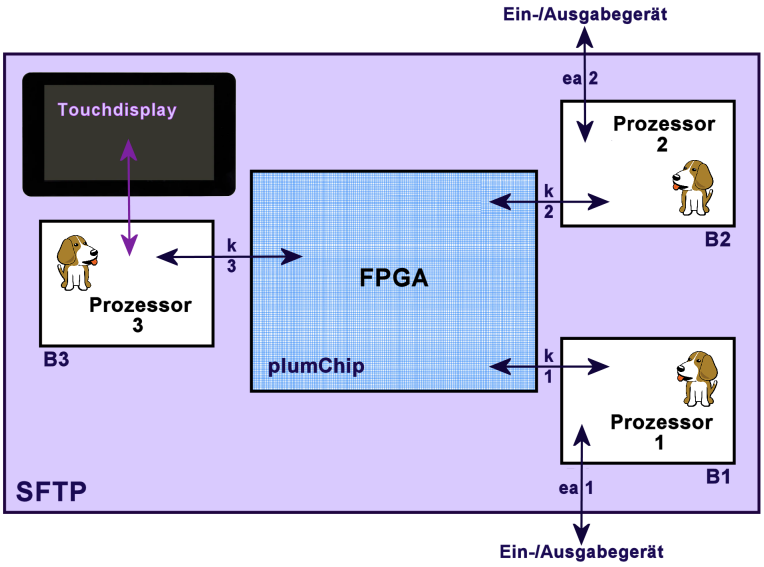


Abbildung 1: Der Zugriff auf den `plumChip` erfolgt wesentlich über den Prozessor *Beagle 3*.

1 Allgemeine Hinweise

Für die drei Prozessoren sind Programme zu schreiben, die den Betrieb der `aPlum` für die gewünschten Anwendungsfälle ermöglichen. Der dafür zur Verfügung stehende Befehlssatz des `plumChip`s ist Gegenstand dieser Beschreibung. Während die Programme für die Prozessoren 1 und 2 als Serviceprogramme zur Abwicklung der angeforderten Ein- und Ausgaben verstanden werden, hat der Prozessor 3 nicht nur den Verschlüsselungsprozess durch den `plumChip` anzuleiten, sondern auch die Benutzerführung abzuarbeiten.

[1] aplum — associative processes linking universal memory, d. i. in assoziative Prozesse einzubindender, vielseitig einsetzbarer Speicherbaustein; aPlum — Gerät, welches einen aplum für kryptografische Zwecke einsetzt; staPlum — stationär zu betreibende Version einer aPlum

[2] SFTP-Board — staPlum-FPGA-Trägerplatine

[3] FPGA — Field Programmable Gate Array, d. i. ein integrierter Schaltkreis, in den digitale, logische Schaltungen geladen werden können

[4] Als Prozessoren sind zurzeit drei `Beagle Bone Black` eingesetzt.

[5] UART — Universal Asynchronous Receiver Transmitter, d. i. Schaltung für digitale serielle Schnittstellen

[6] Auf der SFTP-Platine ist er durch die Aufschrift „B3“ markiert.

Die nachfolgenden Beschreibungen werden durch erklärende C-Programmtexte begleitet. Diese Texte entstammen Testumgebungen, die unter Debian mit dem Kompilierer *gcc (Debian 6.3.0-18+deb9u1) 6.3.0 20170516* erzeugt wurden.[7]

1.1 Bereitschafts-, Erfolgs- und Misserfolgsanzeige

Jeder ausgeführte Befehl wird im Erfolgsfalle mit der Meldung `>ok<` quittiert. Die Ausgaben von Registerinhalten und Matrixinhalt werden mit dem Zeichen „>“ eingeleitet und dem Zeichen „<“ beendet. Befehle, die dem `plumChip` nicht bekannt sind oder die ihn über eine falsche Schnittstelle erreichen, werden vorne mit der Zeichenfolge `>ERR:` gekennzeichnet zurückgemeldet.

1.2 Einrichten der seriellen Schnittstellen

Die seriellen Schnittstellen k_1, k_2 und k_3 (siehe Abbildung 1) lassen sich durch folgendes Unterprogramm konfigurieren, dem man auch die benutzten Parameterwerte entnehmen kann. Es wird insbesondere erwartet, dass `/dev/ttyO4` als UART4-Schnittstelle eingerichtet ist (siehe auch Kapitel 4.2).

```
 1  int OeffneSeriell(void) {
 2    /* Öffnet einen seriellen Port und gibt die zugehörige Dateizugriffs-    */
 3    /* nummer zurück oder den Wert -1 bei einem Fehler.                      */
 4    /* RS232-Parameter: 115200 bps, 8 Datenbits, 1 Stoppbit, no parity,      */
 5    /*                  no handshake                                         */
 6    int fd;                                          /* Dateizugriffsnummer */
 7    struct termios options;                       /* Schnittstellenoptionen */
 8                            /* Port öffnen - read/write, kein "controlling   */
 9                            /* tty" und Status von DCD ignorieren            */
10    fd = open("/dev/ttyO4",O_RDWR | O_NOCTTY | O_NDELAY);
11    if (fd >= 0) {
12      fcntl(fd,F_SETFL,0);
13      if (tcgetattr(fd,&options) != 0) return(-1);
14                                          /* Optionen sichern und bei Pro- */
15      memset(&options,0,sizeof(options));  /* grammende wieder restaurieren */
16      cfsetispeed(&options,B115200);       /* Baudrate setzen */
17      cfsetospeed(&options,B115200);
18      options.c_cflag &= ~PARENB;          /* kein Paritybit */
19      options.c_cflag &= ~CSTOPB;          /* 1 Stoppbit */
20      options.c_cflag &= ~CSIZE;           /* 8 Datenbits */
21      options.c_cflag |= CS8;
22      options.c_cflag |= (CLOCAL | CREAD);  /* CD-Signal ignorieren,         */
23                                           /* Lesen erlauben                */
24      options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG); /* Kein Echo, keine
25                                            Steuerzeichen, keine Interrupts */
26      options.c_iflag = IGNPAR;            /* Parity-Fehler ignorieren */
27      options.c_oflag &= ~OPOST;           /* setze "raw" Input */
28      options.c_cc[VMIN] = 0;              /* warten auf min. 0 Zeichen */
29      options.c_cc[VTIME] = 10;            /* Timeout 1 Sekunde */
30      tcflush(fd,TCIOFLUSH);               /* Puffer leeren */
31      if (tcsetattr(fd,TCSAFLUSH,&options) != 0) return(-1);
32    }
33    return(fd);
34  } /* Ende OeffneSeriell */
```

[7] `GCC` — GNU Compiler Collection, `gcc` — Compiler und Linker für verschiedene Sprachen

1.3 Senden und Empfangen über die seriellen Schnittstellen

Über die seriellen Schnittstellen k_1, k_2 und k_3 versendet und empfängt man Bytes durch Unterprogramme wie zum Beispiel die folgenden beiden:

```
 1  int SendeNachricht(int f, char* Puffer, int anz) {
 2    /* Sendet <anz> Bytes aus dem <Puffer> an den plumChip und liefert die  */
 3    /* Anzahl gesendeter Zeichen zurück. <f> ist die Dateizugriffsnummer    */
 4    /* der seriellen Schnittstelle.                                         */
 5    int gesendet;                              /* Anzahl gesendeter Zeichen */
 6    gesendet = write(f,Puffer,anz);   /* Versand der Zeichen aus dem Puffer */
 7    if (gesendet < 0) {
 8      printf("  +++ Fehler in SendeNachricht: Zeichen nicht gesendet");
 9      return -1;
10    }
11    if (gesendet < anz) {
12      printf("  +++ Fehler in SendeNachricht: Zeichen zum Teil nicht gesendet");
13    }
14    return gesendet;
15  } /* Ende SendeNachricht */
```

Der obige Programmtext bewirkt, dass eine bestimmte Anzahl Zeichen über die serielle Schnittstelle versandt wird. Durch den Einsatz der C-Unterprogramme `write` (siehe oben Zeile 6) und `read` (siehe unten Zeile 9) stellt man sicher, dass auch Sonderzeichen wie NULL als solche übertragen werden und die Verbindung nicht abgebrochen wird.

Der nachfolgende Programmtext legt ankommende Zeichen in einem Lesepuffer ab, bis vom `plumChip` das Endezeichen '<' eintrifft. Das unterstellt, dass dieses Zeichen nur zu diesem Zweck in den zu empfangenden Nachrichten auftaucht.

```
 1  int EmpfangeNachricht(int f, char* Puffer) {
 2    /* Empfängt Bytes vom plumChip in den <Puffer> und liefert deren Anzahl */
 3    /* zurück. <f> ist die Dateizugriffsnummer der seriellen Schnittstelle. */
 4    int  anz;                              /* Anzahl eingelesener Zeichen  */
 5    int  idx;                              /* Index für <Puffer>           */
 6    char c;
 7    idx = 0;
 8    do {
 9      anz = read(f,(void*)&c,1);              /* beliebige Zeichen einlesen */
10      if (anz > 0) {
11        if (c != '<') {
12          Puffer[idx++] = c;
13        }
14      }
15    } while (c != '<');                        /* Endezeichen '<' erreicht */
16    Puffer[idx++] = '<';           /* Endezeichen dem Lesepuffer hinzufügen */
17    if (anz < 0) {
18      printf("  +++ Fehler in EmpfangeNachricht: Einlesefehler");
19    } else {
20      Puffer[idx] = '\0';                          /* String-Ende festlegen */
21    }
22    return idx;                    /* Anzahl gelesener Zeichen zurückmelden */
23  } /* Ende EmpfangeNachricht */
```

1.4 Zurücksetzen des plumChips

Einen **Reset** des plumChip löst man aus, indem man den Kontaktstift AA22 (Leiste C4, Pin #22) auf Masse zieht. Nutzt man dafür den Kontaktstift *GPIO*47[8] eines Beagle Bone Black, wäre das C-programmgesteuert durch folgende Befehle zu erreichen:

```
1  /* Reset ausführen */
2  system("echo out > /sys/class/gpio/gpio47/direction");
3  system("echo 0 > /sys/class/gpio/gpio47/value");
4  warte(1); /* Abwartetakt */
5  system("echo 1 > /sys/class/gpio/gpio47/value");
```

1.5 Kontrollanzeigen

Wählt man beispielsweise den Kontaktstift *GPIO*46 eines Beagle Bone Black, um eine LED[9] als Kontrollanzeige zu betreiben, erreicht man das programmgesteuert durch folgende Befehle:[10]

```
1  /* LED an-/ausschalten */
2  system("echo out > /sys/class/gpio/gpio46/direction");
3  system("echo 1 > /sys/class/gpio/gpio46/value");
4  warte(2); /* Blinkdauer */
5  system("echo 0 > /sys/class/gpio/gpio46/value");
```

2 Zugriff auf die Matrix des plumChips

Im plumChip werden die assoziativen Vorgänge derzeit über eine quadratische Matrix M abgewickelt, die hier als A-Matrix bezeichnet wird und die in den RAM-Blöcken des FPGA untergebracht ist. Da ein einziger RAM-Block dafür zumeist nicht ausreicht,[11] werden mehrere RAM-Blöcke zusammengelegt (siehe Abbildung 2). Als Programmierender braucht man darauf keine Rücksicht zu nehmen, denn der plumChip erledigt den Zugriff so, als hätte man die Matrix im Ganzen vor sich.

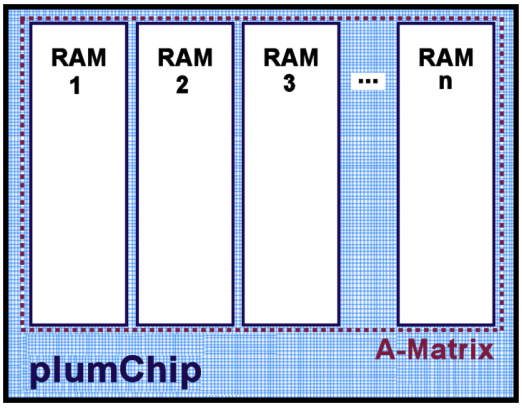


Abbildung 2: Die A-Matrix wird im plumChip aus mehreren RAM-Blöcken zusammengesetzt.

2.1 Löschen der Matrix

Unter dem *Löschen* der Matrix des plumChips wird das Eintragen von Nullen in die Matrix verstanden. Man erreicht dieses programmgesteuert durch den Prozessor 3, indem man von dort dem plumChip das Befehlszeichen 'C' über die Schnittstelle k_3 sendet.

[8] GPIO — general purpose input/output, d. i. ein Allzweck-Ein-/Ausgabe-Kontaktstift (Pin)

[9] LED — light-emitting diode, d.i. eine Leuchtdiode

[10] Polung der LED beachten: Anode an Kontaktstift *GPIO*46, und strombegrenzenden Widerstand von etwa 470 Ω in Reihe schalten, damit die Stromstärke kleiner als die erlaubten 8 mA bleibt!

[11] Im zurzeit eingesetzten FPGA befinden sich RAM-Blöcke der Größe 10 KBit.

```
1  bool MatrixLoeschen(int f) {
2    /* Löscht die Matrix des plumChips. <f> ist die Dateizugriffsnummer der */
3    /* seriellen Schnittstelle.                                             */
4    char tmp[256];                      /* temporärer Lese-/Empfangspuffer */
5    SendeNachricht(f,"C",1);                  /* Befehlszeichen 'C' senden */
6    EmpfangeNachricht(f,tmp);                 /* Empfang von >ok< erwarten */
7    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
8  } /* Ende MatrixLoeschen */
```

Nach dem erfolgreichen Abarbeiten des Befehls, sendet der `plumChip` die Bereitschaftsanzeige `>ok<` an alle Prozessoren, so dass alle von der erfolgreichen Ausführung des Befehls Kenntnis erhalten. Dieses Verhalten zeigt der `plumChip` bei allen nachstehenden Befehlen.

2.2 Zufälliges Befüllen der Matrixspalten zur Hälfte mit Einsen

Um alle **Spalten** der Matrix zur Hälfte zufällig mit Einsen zu belegen, beginnt der `plumChip` im RAM-Block 1 in Spalte 1 (ganz links, siehe Abbildung 2) mit Hilfe seines (Pseudo-) Zufallsgenerators, ein alternierender Stop-and-Go-Generator, solange Positionen auszuwürfeln und dort Einsen hineinzuschreiben, bis die Spalte genau gleich viele gesetzte wie gelöschte Bits enthält, gleich viele Nullen und Einsen. Sind die Spalten eines RAM-Blocks von links nach rechts auf diese Weise versorgt worden, setzt der `plumChip` mit dem nächsten RAM-Block fort, immer streng in der Reihenfolge von links nach rechts.

```
1  bool MatrixZufSpalten50Fuellen(int f) {
2    /* Füllt die Spalten der Matrix des plumChips zu jeweils 50 Prozent zu- */
3    /* fällig mit Einsen. <f> ist die Dateizugriffsnummer der seriellen     */
4    /* Schnittstelle.                                                       */
5    char tmp[256];                      /* temporärer Lese-/Empfangspuffer */
6    SendeNachricht(f,"z",1);                  /* Befehlszeichen 'z' senden */
7    EmpfangeNachricht(f,tmp);                 /* Empfang von >ok< erwarten */
8    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
9  } /* Ende MatrixZufSpalten50Fuellen */
```

Hinweis: Man achte darauf, vor dem zufälligen Befüllen der Matrix diese zu **löschen**, damit nicht womöglich in den Spalten bereits vorhandene Einsen das Erreichen des Ziels verhindern! Merke: *„Zuerst Cee, dann Zett!“*.

2.3 Zufälliges Befüllen der Matrixzeilen zur Hälfte mit Einsen

Um alle **Zeilen** der Matrix zur Hälfte zufällig mit Einsen zu belegen, beginnt der `plumChip` in der ersten Zeile durch alle RAM-Blöcke hindurch mit Hilfe seines (Pseudo-) Zufallsgenerators, solange Positionen auszuwürfeln und dort Einsen hineinzuschreiben, bis die Zeile genau gleich viele gesetzte wie gelöschte Bits enthält, gleich viele Nullen und Einsen. Dann setzt der `plumChip` mit der nächsten Zeile fort, immer streng in der Reihenfolge von oben nach unten (siehe Abbildung 2).

```
1  bool MatrixZufZeilen50Fuellen(int f) {
2    /* Füllt die Zeilen der Matrix des plumChips zu jeweils 50 Prozent zu-  */
3    /* fällig mit Einsen. <f> ist die Dateizugriffsnummer der seriellen     */
4    /* Schnittstelle.                                                       */
5    char tmp[256];                      /* temporärer Lese-/Empfangspuffer */
6    SendeNachricht(f,"y",1);                  /* Befehlszeichen 'y' senden */
7    EmpfangeNachricht(f,tmp);                 /* Empfang von >ok< erwarten */
8    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
9  } /* Ende MatrixZufZeilen50Fuellen */
```

2.4 Weitere Matrixbefehle

Für die kryptografischen Verfahren der `aPlum` reichen die in den Kapiteln 2.1 bis 2.2 beschriebenen Matrixbefehle aus. Zum Testen und zur Fehlersuche können zurzeit weitere Matrixbefehle durch den `plumChip` ausgeführt werden. Die Tabelle 6.1 zählt diese auf. Nach dem Muster der Unterprogramme in den Kapiteln 2.1 bis 2.2 lassen sich diese aufrufen, indem man vor allem die Programmtextzeile 6 entsprechend anpasst.

3 Zugriff auf die Register des `plumChips`

Neben der Matrix enthält der `plumChip` ein Lernantwortregister L und ein Frageregister Q, wie in Abbildung 3 dargestellt. Im allgemeinen Fall dienen diese beiden Register der Aufnahme eines Frage-Antwort-Paares, welches in die Matrix durch die L- oder K-Lernregel eingetragen werden soll (siehe [BentzDierks13], Kapitel 2.5.2). Für kryptografische Zwecke werden die beiden Register anders eingesetzt.

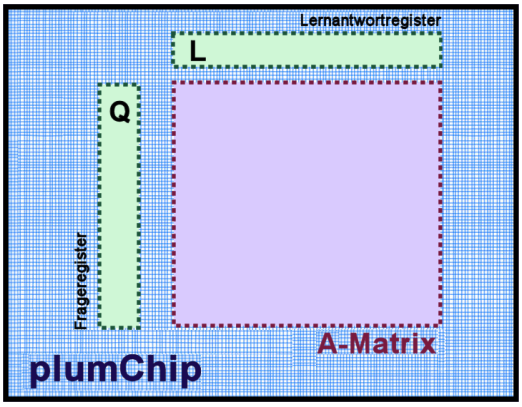


Abbildung 3: Der `plumChip` enthält neben der A-Matrix ein Lernantwort- L und ein Frageregister Q.

Das Frageregister dient beim Verschlüsseln von Daten der Ablage des für den jeweiligen Block genutzten Schlüssels. Das Lernantwortregister enthält den zu ver- oder entschlüsselnden Datenblock. Aus der Vielzahl der Registerbefehle, die im `plumChip` über das notwendige Maß hinaus zu Prüf- und Kontrollzwecken implementiert wurden (siehe Tabelle 6.2), werden hier nur die kryptografisch genutzten Befehle beschrieben. Nichtsdestotrotz lassen sich die weiteren Registerbefehle nach dem Muster der folgenden Programmtextbeispiele durch den Programmierenden ausführen. Bei allen Befehlen meldet der `plumChip` nach deren erfolgreicher Ausführung die Bereitschaftsanzeige `>ok<` über alle drei seriellen Schnittstellen k_1, k_2 und k_3 zurück.

3.1 Zufälliges Befüllen des Frageregisters zur Hälfte mit Einsen

Um eine Frage zu erzeugen, die durch Zufall bestimmt genau gleich viele Nullen wie Einsen enthält, ruft man den `plumChip`-Befehl 'x' auf. Das nachfolgende Beispiel eines Unterprogrammtextes leistet diese Aufgabe.

```
1  bool FrageZuf50Fuellen(int f) {
2    /* Füllt das Fragetupel des plumChips zufällig zu 50 Prozent mit        */
3    /* Einsen. <f> ist die Dateizugriffsnummer der seriellen Schnittstelle. */
4    char tmp[256];                        /* temporärer Lese-/Empfangspuffer */
5    SendeNachricht(f,"x",1);                    /* Befehlszeichen 'x' senden */
6    EmpfangeNachricht(f,tmp);                   /* Empfang von >ok< erwarten */
7    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
8  } /* FrageZuf50Fuellen */
```

Das Frageregister muss vor Aufruf dieses Befehls nicht eigens gelöscht werden. Das erledigt der `plumChip` in seinem Initialisierungsteil des Befehls. Anschließend werden mit Hilfe des im `plumChip` eingebauten (Pseudo-) Zufallsgenerators die Bits zufällig gesetzt, wie in Kapitel 2.2 für die Matrixspalten bereits beschrieben.

3.2 Eingabe ins Lernantwortregister

Das Eintragen von Daten ins Lernantwortregister lässt sich von allen drei Prozessoren aus bewerkstelligen. Möchte der Programmierende die Daten von Prozessor 1 über k_1 zum `plumChip` leiten, dann wählt der sie Variante ',l'. Sollen die Daten vom Prozessor 2 über k_2 empfangen werden, wählt er '.l' und sonst ':l'. Merke: *„Eingabe: Komma — Beagle 1, Punkt — Beagle 2, Doppelpunkt — Beagle 3“*.

```
 1  bool LernantwortEingeben(int f, char* Antwort) {
 2    /* Trägt <Antwort> in das Lernantwortregister des plumChips ein.        */
 3    /* <f> ist die Dateizugriffsnummer der seriellen Schnittstelle.         */
 4    /* Es wird als globale Variable <bgroesse> mit der Größe der Antwort    */
 5    /* in Bytes erwartet.                                                   */
 6    char tmp[256];                        /* temporärer Lese-/Empfangspuffer */
 7    SendeNachricht(f,":l",2);                  /* Befehlszeichen ':l' senden */
 8    SendeNachricht(f,Antwort,bgroesse);                  /* <Antwort> senden */
 9    EmpfangeNachricht(f,tmp);                   /* Empfang von >ok< erwarten */
10    return (strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
11  } /* Ende LernantwortEingeben */
```

Eine Variante der Eingabe eines Datenblocks ins Lernantwortregister erlauben die Befehle ',L', '.L' und ':L'. Diese ermöglichen die Eingabe der Bytes nicht in binärer Form, sondern durch die Zeichen '0' und '1' (siehe Kapitel 6.2.2).

3.3 Ausgabe des Lernantwortregisters

Die Ausgabe des Inhalts des Lernantwortregisters lässt sich zu allen drei Prozessoren hin ausführen. Möchte der Programmierende die Daten vom `plumChip` zu Prozessor 1 über k_1 leiten, dann wählt er die Variante '<t'. Sollen die Daten vom `plumChip` zum Prozessor 2 über k_2 gelangen, wählt er '>t' und sonst '=t'. Merke: *„Ausgabe: kleiner — Beagle 1, größer — Beagle 2, gleich — Beagle 3“*.

```
 1  bool LernantwortAusgeben(int f) {
 2    /* Lässt den Inhalt des Lernantwortregisters des plumChips zum Beagle 2 */
 3    /* ausgeben. <f> ist die Dateizugriffsnummer der seriellen Schnitt-     */
 4    /* stelle.                                                              */
 5    char tmp[256];                        /* temporärer Lese-/Empfangspuffer */
 6    SendeNachricht(f,">t",2);              /* Lernantwort an Beagle 2 senden */
 7    EmpfangeNachricht(f,tmp);                   /* Empfang von >ok< erwarten */
 8    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
 9  } /* Ende LernantwortAusgeben */
```

Der Empfänger des Inhalts des Lernantwortregisters muss die Ausgabe des `plumChip`s zur richtigen Zeit erwarten und entgegennehmen, wobei ihm die Bereitschaftsanzeige `>ok<` hilft, die der `plumChip` nach erfolgreicher Abarbeitung eines Befehls jeweils allen Prozessoren sendet.

Die Ausgabe des Lernantwortregisterinhalts durch die Zeichen '0' und '1' erreicht der Programmierende durch die Befehle '<T', '>T' und '=T' (siehe Kapitel 6.2.2).

3.4 Frage- und Lernantwortregister mit XOR verknüpfen

Die Verschlüsselung eines Datenblocks, der im Lernantwortregister liegt, erfolgt durch eine XOR-Verknüpfung mit dem Schlüssel, der sich im Frageregister befindet.

```
 1  bool XORAusfuehren(int f) {
 2    /* Verknüpft den Inhalt des Frage- mit dem des Lernantwortregisters des */
 3    /* plumChips und trägt das Ergebnis ins Lernantwortregister ein.        */
 4    /* <f> ist die Dateizugriffsnummer der seriellen Schnittstelle.         */
 5    char tmp[256];                        /* temporärer Lese-/Empfangspuffer */
 6    SendeNachricht(f,"o",1);                    /* Befehlszeichen 'o' senden */
 7    EmpfangeNachricht(f,tmp);                   /* Empfang von >ok< abwarten */
 8    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
 9  } /* Ende XORAusfuehren */
```

Damit sich im Frageregister ein geeigneter Schlüssel befindet, ist zuvor entweder der Befehl 'x' (siehe Kapitel 3.1) zu geben oder es sind ein oder mehrere Schlüsselerzeugungsbefehle (siehe Kapitel 3.5) durch die `plumChip`-Befehle 'n' (MINA), 'N' (LSB) oder 'i' (Sigma) aufzurufen.

3.5 Schlüssel erzeugen

Im `plumChip` befinden sich Schaltungen für drei verschiedene Verfahren zur Schlüsselerzeugung, das MINA-, LSB- und Sigma-Verfahren.[12] Durch diese Schaltungen wird aus dem bisherigen Inhalt des Frageregisters mit Hilfe der A-Matrix der nächste Inhalt des Frageregisters gebildet. Dadurch entsteht eine Iteriertenfolge,[13] ein Schlüsselstrom im Sinne eines Schlüsselstromgenerators für Blockschlüssel (vgl. [Schneier15], S. 369ff). Die Schlüsselerzeugungsverfahren lassen sich auch miteinander kombinieren oder mehrfach ausführen, wofür der folgende Unterprogrammtext Beispiele gibt.

```
 1  bool NaechsteIterierteBilden(int f) {
 2    /* Veranlasst den plumChip, den nächsten Schlüssel ins Frageregister zu */
 3    /* schreiben. <f> ist die Dateizugriffsnummer der seriellen Schnitt-    */
 4    /* stelle. In der globalen Variablen <vv> wird das zu verwendende       */
 5    /* Verschlüsselungsverfahren erwartet: M für MINA, L für LSB, i für     */
 6    /* Sigma, ML für MINA/LSB, MM für MINA/MINA, LM für LSB/MINA u.s.f.     */
 4    char tmp[256];                        /* temporärer Lese-/Empfangspuffer */
 5    if (strcmp(vv,"M") == 0)
 6      SendeNachricht(f,"n",1);
 7    else if (strcmp(vv,"L") == 0)
 8      SendeNachricht(f,"N",1);
 9    else if (strcmp(vv,"S") == 0)
10      SendeNachricht(f,"i",1);
11    else if (strcmp(vv,"LM") == 0) {
12      SendeNachricht(f,"N",1);
13      SendeNachricht(f,"n",1);
14    } else if (strcmp(vv,"ML") == 0) {
15      SendeNachricht(f,"n",1);
16      SendeNachricht(f,"N",1);
17    } else if (strcmp(vv,"MM") == 0) {
18      SendeNachricht(f,"n",1);
19      SendeNachricht(f,"n",1);
20    } else {
21      printf("  +++ Verschlüsselungsverfahren unbekannt: %s",vv);
22      exit(1);
23    }
24    EmpfangeNachricht(f,tmp); /* Kumulationssignale abfangen */
25    EmpfangeNachricht(f,tmp); /* >ok< abfangen */
```

[12] siehe [Dierks18-7] bezüglich der Unterschiede der Verfahren

[13] Iterierte — Zwischenergebnisse ein und desselben Verfahrens, das wiederholt angewandt wird; hier: Es entsteht dadurch eine Folge an Blockschlüsseln.

```
26    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
27  } /* Ende NaechsteIterierteBilden */
```

Der Kommentar in Zeile 24 des obigen Programmtexts zu „Kumulationssignalen“ bezieht sich auf die Ausgabe von Kontrollzeichen '■' des `plumChips`, mit denen die Effektivität das Verschlüsselungsverfahrens überprüft werden kann (siehe Kapitel 6.3).

3.6 Salz holen

Damit ein und dieselben Daten trotz gleichen Benutzers oder gleichen Passworts immer wieder mit einer anderen Folge an Schlüsseln verarbeitet werden kann, wird in den Beginn des Verschlüsselungsprozesses ein Salz[14] eingebracht, das aus „echtem“ Zufall entsteht. Am Verschlüsselungsergebnis ist somit nicht ablesbar, welche Dateien oder andere Daten das Verschlüsselungsverfahren durchlaufen haben.

```
 1  bool SalzHolen(int f) {
 2    /* Holt aus dem Salzgenerator des plumChips ein 128 Bit langes Salz und */
 3    /* legt es in der globalen Variablen <salz> ab. <f> ist die Datei-      */
 4    /* zugriffsnummer der seriellen Schnittstelle.                          */
 5    char tmp[256];                       /* temporärer Lese-/Empfangspuffer */
 6    SendeNachricht(f,"s",1);                   /* Befehlszeichen 's' senden */
 7    EmpfangeNachricht(f,salz);                           /* Salz empfangen */
 8    EmpfangeNachricht(f,tmp);                  /* Empfang von >ok< abwarten */
 9    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
10  } /* Ende SalzHolen */
```

Der Salzgenerator besteht derzeit aus einem gesonderten, unabhängigen alternierenden Stop-and-Go-Generator (vgl. [Schneier15], S. 383), der unentwegt „würfelt“ solange am `plumChip` eine Versorgungsspannung anliegt.

3.7 Weitere Registerbefehle

Auch wenn die vorstehend beschriebenen Befehle dem Programmierenden zur Erstellung der Software für eine `aPlum` bereits genügen dürften, so listet die Tabelle 6.2 eine Reihe weiterer Befehle auf, die vornehmlich Test- und Kontrollaufgaben dienten.

4 Verwaltungsbefehle

4.1 Kennung des `plumChips` einlesen

Der `plumChip` liefert bei Eingabe des Befehls 'I' eine Zeichenkette der Form `>imbit.net,C5,SFTP5,v1.8.1,0128<` an alle drei Prozessoren, der man als Programmierer, wie in folgendem Unterprogramm angedeutet, Informationen über die Entwicklungsplatine, den Befehlssatz und die Größe der A-Matrix.

```
 1  void KennungAuswerten(int f) {
 2    /* Liest die Kennung des plumChips ein und wertet diese aus. <f> ist    */
 3    /* die Dateizugriffsnummer der seriellen Schnittstelle. Die Matrix- und */
 4    /* die sich daraus ergebende Blockgröße wird in den globalen Variablen  */
 5    /* <mgroesse> und <bgroesse> zurückgeliefert.                           */
 6    char id[50];                                            /* Lesepuffer */
 7    char trenn[] = ",";                                   /* Trennzeichen */
 8    char *teil;                                             /* Teilstring */
```

[14] In der Kryptologie wird eine zufällig gewählte Zeichenfolge als ein *Salt* bezeichnet, wenn dadurch die Entropie einer Eingabe erhöht werden soll. Verwendet man eine solche zufällige Zeichenfolge hingegen wie oben beschrieben mit dem Ziel, einen zu verschlüsselnden Text nie auf dasselbe Ergebnis abzubilden, nennt man diese eine *Nonce*. Für die `aPlum` benutzen wir in der vorliegenden Beschreibung den Begriff *Salz* für einen 128 Bit langen Schlüssel, mit dem wir beides erreichen wollen.

```
 9    SendeNachricht(f,"I",1);                    /* Befehlszeichen 'I' senden */
10    EmpfangeNachricht(f,id);                           /* Kennung empfangen */
11    teil = strtok(id,trenn);   /* >imbit.net */
12    teil = strtok(NULL,trenn); /* C5     */
13    teil = strtok(NULL,trenn); /* SFTP5  */
14    teil = strtok(NULL,trenn); /* v1.8.1 */
15    teil = strtok(NULL,trenn); /* 0128<  */
16    teil[4] = '\0';                      /* Ende des Teilstrings markieren */
17    mgroesse = atoi(teil);                        /* Matrixgröße bestimmen */
18    bgroesse = mgroesse / 8;                       /* Blockgröße berechnen */
19  } /* Ende KennungAuswerten */
```

Da für einige Programmteile die korrekten Werte der Matrix- und Blockgröße entscheidend sind, sollte die Abfrage der Kennung am Beginn jedes Programms erfolgen, auch in den Programmen der `aPlum`, die auf den Prozessoren 1 und 2 laufen. Daher gehört der Befehl 'I' zu den wenigen Befehlen, den auch die Prozessoren 1 und 2 aufrufen dürfen, ohne einen Fehler `>ERR:` zurückgemeldet zu bekommen.

4.2 Sekundensignal ein- und ausschalten

Zur Prüfung der UART-Verbindungen k_1, k_2 und k_3 gibt der `plumChip` nach einem Neustart einmal pro Sekunde die Bereitschaftsanzeige `>ok<` an alle drei Prozessoren aus. Mit dem Befehl 'S' lässt sich diese Anzeige ein- und ausschalten. Dieser Befehl darf dem `plumChip` von allen drei Prozessoren aus gegeben werden.

```
 1  bool SekundensignalAnAus(int f) {
 2    /* Schaltet das Sekundensignal des plumChip an und aus.                */
 3    char tmp[256];                       /* temporärer Lese-/Empfangspuffer */
 4    SendeNachricht(f,"S",1);                   /* Befehlszeichen 'S' senden */
 5    EmpfangeNachricht(f,tmp);                  /* Empfang von >ok< abwarten */
 6    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
 7  } /* Ende SekundensignalAnAus */
```

Durch den Befehl ' ' ' (Apostroph) lässt sich das Sekundensignal sofort ausschalten, egal ob es vorher schon ausgeschaltet war oder nicht.

```
 1  bool SekundensignalAus(int f) {
 2    /* Schaltet das Sekundensignal des plumChip aus.                       */
 3    char tmp[256];                       /* temporärer Lese-/Empfangspuffer */
 4    SendeNachricht(f,"'",1);                   /* Befehlszeichen ''' senden */
 5    EmpfangeNachricht(f,tmp);                  /* Empfang von >ok< abwarten */
 6    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
 7  } /* Ende SekundensignalAus */
```

Dieser Apostroph-Befehl lässt sich nur vom Prozessor 3 aus geben.

Hinweis: Sollte kein Sekundensignal bei den Prozessoren eintreffen, sind die UART-Schnittstellen auf korrekte Verkabelung zu überprüfen und im Falle, dass *Beagles* als Prozessoren eingesetzt sind, in der Systemdatei `uEnv.txt` zu kontrollieren, ob `/dev/ttyO4` als UART4-Schnittstelle beim Booten eingerichtet wird.

4.3 Zufallsgenerator initialisieren

Nach der Beschreibung im [Schneier15] besteht der als Pseudozufallsgenerator im `plumChip` verwendete alternierende Stop-and-Go-Generator aus drei Schieberegistern (siehe Abbildung 4), deren Startzustand durch eine **Saat** festgelegt wird, die in der vorliegenden Version 30 Bytes lang ist.

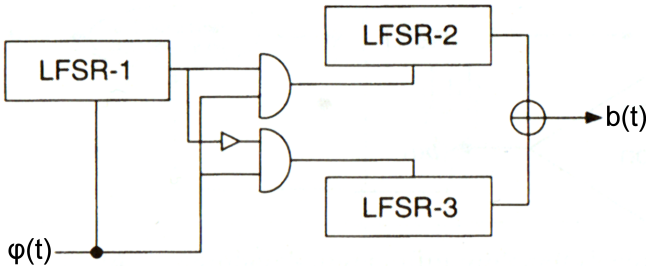


Abbildung 4: Alternierender Stop-and-Go-Generator

Die Verteilung der 30 Bytes auf die drei LFSR[15] leistet der `plumChip` bei Ausführung des Befehls 'D'.[16]

```
 1  bool SchluesselEingeben(int f, char* key) {
 2    /* Fragt nach dem Schlüssel, der als Saat für den LFSR-Zufallszahlen-   */
 3    /* generator vom FPGA genutzt wird.                                     */
 4    char tmp[256];                        /* temporärer Lese-/Empfangspuffer */
 5    SendeNachricht(f,"D",1);                    /* Befehlszeichen 'D' senden */
 6    SendeNachricht(f,key,30);             /* Saatschlüssel senden: 30 Bytes */
 7    EmpfangeNachricht(f,tmp);                   /* Empfang von >ok< abwarten */
 8    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten */
 9  } /* Ende SchluesselEingeben */
```

Die erfolgreiche Durchführung des Verteilens der Saat auf die drei Schieberegister meldet der `plumChip` durch die Ausgabe der Bereitschaftsanzeige `>ok<` an alle drei Prozessoren.

Der Pseudozufallsgenerator des `plumChip` dient nicht der Erzeugung eines Schlüsselstroms zum Verschlüsseln von Datenblöcken, sondern lediglich der Versorgung von A-Matrix (Befehle 'y' und 'z', siehe Kapitel 2.3 und 2.2) und gegebenenfalls des Frageregisters (Befehl 'x', siehe Kapitel 3.1).

4.4 Inhalt des Zufallsgenerators anzeigen

Zur Anzeige des aktuellen Inhalts des Pseudozufallsgenerators steht der 'd'-Befehl zur Verfügung, durch den der `plumChip` über die Leitung k_3 die Bytes sendet, die den Zustand des Generators beschreiben.

```
 1  bool SchluesselHolen(int f, char* key) {
 2    /* Holt aus dem Zufallszahlengenerator des plumChips die dreißig Bytes, */
 3    /* die seinen aktuellen Zustand darstellen. In <key> werden diese Bytes */
 4    /* geliefert. <f> ist die Dateizugriffsnummer der seriellen Schnitt-    */
 5    /* stelle.                                                              */
 5    char tmp[256];                        /* temporärer Lese-/Empfangspuffer */
 6    SendeNachricht(f,"d",1);                    /* Befehlszeichen 'd' senden */
 7    EmpfangeNachricht(f,key);                     /* Saatschlüssel empfangen */
 8    EmpfangeNachricht(f,tmp);                   /* Empfang von >ok< abwarten */
 9    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
10  } /* Ende SchluesselHolen */
```

Die Ausgabe des Inhalts des Zufallsgenerators hat die Gestalt `>A.86.z.|bc9...0|..HG4.>ok<`, wobei an die Stelle nicht darstellbarer Zeichen ein Punkt tritt. Es wird der Inhalt jedes der drei LFSR, voneinander abgetrennt durch senkrechte Striche '|' angezeigt.

[15] LFSR — Linear Feedback Shift Register, d. i. ein lineares Schieberegister

[16] Zum Einsatz kommen hier drei Fibonacci-LFSR, siehe dazu auch [Dierks19-7]. Daher darf die Saat nicht nur aus Nullen bestehen.

5 Kommunikationsbefehle

Allen Kommunikationsbefehlen des `plumChip` ist gemeinsam, dass vorher festgelegt sein muss, wie viele Bytes von einem Prozessor zu einem anderen durch den `plumChip` über die Verbindungen k_1, k_2 oder k_3 transportiert werden sollen. Ist die vorher angegebene Anzahl an Bytes erreicht, sendet der `plumChip` die Bereitschaftsanzeige `>ok<` an alle Prozessoren aus. Zu beachten ist ferner, dass alle Kommunikationsbefehle durch den Prozessor 3 ausgelöst werden, selbst wenn zum Beispiel der Prozessor 2 dem Prozessor 1 etwas mitzuteilen hat, der Prozessor 3 am Nachrichtentransfer also gar nicht beteiligt wäre.

5.1 Nachrichtenlänge festlegen

Die gewünschte Nachrichtenlänge wird dem `plumChip` in Form eines Bytes mitgeteilt, dessen numerischer Wert die Länge angibt. Die Länge einer Nachricht darf daher nicht größer als 255 sein. Der folgende Unterprogrammtext ist ein Muster für dieses Vorgehen.

```
 1  bool LaengeSenden(int f, int lng) {
 2    /* Legt fest, wie lang die zwischen den Prozessoren auszutauschenden    */
 3    /* Nachrichten sind durch <lng> in Bytes.                               */
 4    /* <f> ist die Dateizugriffsnummer der seriellen Schnittstelle.         */
 5    char c;                                          /* Länge der Nachricht */
 6    char tmp[256];                       /* temporärer Lese-/Empfangspuffer */
 7    SendeNachricht(f,"w",1);                   /* Befehlszeichen 'w' senden */
 8    c = (char)lng;                  /* Länge als Zeichen zwischen 0 und 255 */
 9    SendeNachricht(f,&c,1);                                /* Länge senden */
10    EmpfangeNachricht(f,tmp);                  /* Empfang von >ok< abwarten */
11    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
12  } /* Ende LaengeSenden */
```

5.2 Nachrichten an die Prozessoren senden

Die Übermittlung von Nachrichten vom Prozessor 3 (*Beagle* 3) an die beiden anderen Prozessoren besteht aus zwei Schritten: zunächst wird dem `plumChip` die Länge der Nachricht mitgeteilt und danach die Nachricht selbst. Das Beispiel in Kapitel 5.2.1 verdeutlicht dieses.

Erwartet Prozessor 3 hingegen von den Prozessoren 1 oder 2 eine Nachricht, so meldet er sich bei diesen mit der Aufforderung `>go<`. Derjenige Prozessor, der auf diese Weise aktiviert wird, sendet dann entweder die Nachrichtenlänge und die Nachricht oder nur die Nachricht, wenn dessen Länge bereits feststeht und durch den Prozessor 3 dem `plumChip` bereits mitgeteilt wurde. Kapitel 5.2.2 zeigt das beispielgebend.

5.2.1 Prozessor 3 an Prozessor 1 oder 2

Der Befehl '1' sorgt für die Übertragung einer Nachricht an den Prozessor 1. Soll dem Prozessor 2 etwas gesandt werden, nimmt man den Befehl '2', ändert also in folgendem Unterprogrammtext die Zeile 10 entsprechend.

```
 1  bool NachrichtAnBeagle1(int f, char* Nachricht) {
 2    /* Sendet <Nachricht> an den Beagle 1 mit Hilfe der Befehle 'w' und '1'.*/
 3    /* <f> ist die Dateizugriffsnummer der seriellen Schnittstelle.         */
 4    char c;                                          /* Länge der Nachricht */
 5    char tmp[256];                       /* temporärer Lese-/Empfangspuffer */
 6    SendeNachricht(f,"w",1);                   /* Befehlszeichen 'w' senden */
 7    c = (char)strlen(Nachricht);    /* Länge als Zeichen zwischen 0 und 255 */
 8    SendeNachricht(f,&c,1);                                /* Länge senden */
```

```
 9    EmpfangeNachricht(f,tmp);                  /* Empfang von >ok< abwarten */
10    SendeNachricht(f,"1",1);                        /* Beagle 3 an Beagle 1 */
11    SendeNachricht(f,Nachricht,strlen(Nachricht));      /* Nachricht senden */
12    EmpfangeNachricht(f,tmp);                  /* Empfang von >ok< abwarten */
13    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
14  } /* NachrichtAnBeagle1 */
```

5.2.2 Prozessor 1 an Prozessor 2 oder 3

Nehmen wir an, dass Prozessor 3 vom Prozessor 1 eine Nachricht einer bestimmten Länge erwartet. Dann nimmt der folgende Unterprogrammetext die nötigen Vorbereitungen vor.

```
 1  bool NachrichtVonBeagle1(int f, int lng) {
 2    /* Sendet an den Beagle 1 die Aufforderung, eine Nachricht an den       */
 3    /* Beagle 3 zu schicken. Es wird eine Nachricht der Länge <lng> erwar-  */
 4    /* tet. <f> ist die Dateizugriffsnummer der seriellen Schnittstelle.    */
 5    char c;                                          /* Länge der Nachricht */
 6    char tmp[256];                       /* temporärer Lese-/Empfangspuffer */
 7    SendeNachricht(f,"w",1);                   /* Befehlszeichen 'w' senden */
 8    c = (char)lng;                  /* Länge als Zeichen zwischen 0 und 255 */
 9    SendeNachricht(f,&c,1);                                /* Länge senden */
10    EmpfangeNachricht(f,tmp);                  /* Empfang von >ok< abwarten */
11    SendeNachricht(f,"3",1);                        /* Beagle 1 an Beagle 3 */
12    EmpfangeNachricht(f,tmp);                  /* Empfang von >ok< abwarten */
13    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
14  } /* NachrichtVonBeagle1 */
```

Im Programm, welches auf dem Prozessor 1 läuft, müsste nach dem Eintreffen des Signals `>go<` entweder die Nachricht mit der bereits eingerichteten Länge gesendet werden oder erst die Länge der Nachricht (als ein Byte) und dann die Nachricht selbst übermittelt werden, je nach Absicht des Programmierenden.

Sollte der Programmierende veranlassen wollen, dass Prozessor 1 an Prozessor 2 eine Nachricht sendet, dann schickt Prozessor 3 an den `plumChip` den Befehl '4'. Die Zeile 11 in obigem Programmtext wäre dazu zu ändern. Der Prozessor 1 erhält ein `>go<` und schickt dann die Nachricht, die der Prozessor 2 erwartet.

5.2.3 Prozessor 2 an Prozessor 1 oder 3

Nach dem gleichen Schema wie im Programmtextbeispiel des Kapitels 5.2.2 werden Nachrichten vom Prozessor 2 aus versandt.

```
 1  bool NachrichtVonBeagle2(int f, int lng) {
 2    /* Sendet an den Beagle 2 die Aufforderung, eine Nachricht an den       */
 3    /* Beagle 3 zu schicken. Es wird eine Nachricht der Länge <lng> erwar-  */
 4    /* tet. <f> ist die Dateizugriffsnummer der seriellen Schnittstelle.    */
 5    char c;                                          /* Länge der Nachricht */
 6    char tmp[256];                       /* temporärer Lese-/Empfangspuffer */
 7    SendeNachricht(f,"w",1);                   /* Befehlszeichen 'w' senden */
 8    c = (char)lng;                  /* Länge als Zeichen zwischen 0 und 255 */
 9    SendeNachricht(f,&c,1);                                /* Länge senden */
10    EmpfangeNachricht(f,tmp);                  /* Empfang von >ok< abwarten */
11    SendeNachricht(f,"5",1);                        /* Beagle 2 an Beagle 3 */
12    EmpfangeNachricht(f,tmp);                  /* Empfang von >ok< abwarten */
13    return(strcmp(">ok<",tmp) == 0); /* >ok< erhalten? */
14  } /* NachrichtVonBeagle2 */
```

Das Programm auf dem Prozessor 2 wartet auf das Signal `>go<` und reagiert dann wie bereits im Kapitel 5.2.2 beschrieben.

Soll eine Nachricht von Prozessor 2 an den Prozessor 1 geschickt werden, sendet Prozessor 3 an den `plumChip` den Befehl '6'. Dazu wäre die Zeile 11 obigen Beispieltextes zu ändern. Nachdem der Prozessor 2 daraufhin die Aufforderung `>go<` erhalten hat, sendet er die Nachricht an den `plumChip` ab, die dieser an den Prozessor 1 weiterleitet, der die Nachricht bereits erwartet.

5.3 Ein- und Ausgaben der Prozessoren 1 und 2

Um über die Verbindungen ea_1 und ea_2 mit der Außenwelt zu kommunizieren, nutzt der Programmierende die dafür üblichen Ein- und Ausgabebefehle der C-Sprachumgebung. Man achte dabei darauf, keine C-Unterprogramme zu nutzen, die ein NULL-Zeichen als Zeichenkettenendezeichen interpretieren und dann den Datentransfer vorzeitig abbrechen. Mit `fread` und `fwrite` befindet man sich bei der Dateiein- und -ausgabe auf der sicheren Seite, wie folgendes Beispiel eines Unterprogramms zeigt, das auf dem Prozessor 1 (*Beagle* 1) ausgeführt wird.

```
 1  void DateibloeckeVerarbeiten(int f) {
 2    /* Liest eine Datei vom USB-Stick blockweise ein und speichert sie ver- */
 3    /* beziehungsweise entschlüsselt wieder auf dem USB-Stick ab. <f> ist   */
 4    /* die Dateizugriffsnummer der seriellen Schnittstelle.                 */
 2    FILE* Quelle;                                          /* Quelldatei */
 3    FILE* Ziel;                                             /* Zieldatei */
 4    char  Block[bgroesse];                                 /* Dateiblock */
 5    char  kcolB[bgroesse+1];             /* ver-/entschlüsselter Dateiblock */
 6    char  QPfad[256];                                   /* Quelldateipfad */
 7    char  ZPfad[256];                                    /* Zieldateipfad */
 8    char  tmp[256];                      /* temporärer Lese-/Empfangspuffer */
 9    int   i;                                     /* Zähler der Blockanzahl */
10    int   sz;                              /* Größe der Quelldatei in Bytes */
11    bool  ende;                    /* Ist das Ende der Quelldatei erreicht? */
12    bool  am3;                              /* Ist die Datei verschlüsselt? */
13
14    strcpy(QPfad,"/media/usb-drive/");              /* Dateinamen festlegen */
15    strcat(QPfad,liste[dsel-1]);
16    strcpy(ZPfad,"/media/usb-drive/");
17    strcat(ZPfad,liste[dsel-1]);
18    am3 = strstr(ZPfad,".am3") != NULL;
19    if (am3) { /* AM3-Datei entdeckt */
20      strcat(ZPfad,".orig");
21    } else {
22      strcat(ZPfad,".am3");
23    }
24    Quelle = fopen(QPfad,"r");                          /* Dateien öffnen */
25    Ziel = fopen(ZPfad,"w");
27    if (Quelle != NULL) {
28      fseek(Quelle,0L,SEEK_END);
29      sz = ftell(Quelle);                        /* Quelldateigröße merken */
30      fseek(Quelle,0L,SEEK_SET);
31      ende = sz == 0; /* Quelle ist leer */
33      i = 0; /* Zähler der Dateiblöcke */
34      while (!ende) {                           /* Dateiblöcke verarbeiten */
35        fread(&Block,bgroesse,1,Quelle);                  /* Block einlesen */
36        i++;
```

```
37        do {                    /* auf das Block-Signal vom Beagle 3 warten */
38          EmpfangeNachricht(f,tmp);
39        } while (strcmp(tmp,"Block>ok<") != 0);
40        if (!LernantwortEingeben(f,Block)) {      /* Block ins Lernregister */
41          printf("  +++ Lernantwort nicht eingegeben\n");
42        }
43        NachrichtAnBeagle3(f,"Code");    /* verschlüsselten Block anfordern */
44        do {                    /* auf das kcolB-Signal vom Beagle 3 warten */
45          EmpfangeNachricht(f,tmp);
46        } while (strcmp(tmp,"kcolB>ok<") != 0);
47        for (int k=0; k < bgroesse+2; k++) { /* verschlüsselten Block lesen */
48          read(f,tmp+k,1);
49        }
50        for (int k=1; k <= bgroesse; k++) {      /* Anfangs- und Endzeichen */
51          kcolB[k-1] = tmp[k];                   /* weglassen               */
52        }
53        kcolB[bgroesse] = '\0';                     /* Endzeichen eintragen */
54        EmpfangeNachricht(f,tmp);              /* Empfang von >ok< abwarten */
55        if (i*bgroesse < sz) {               /* Ist das Dateiende erreicht? */
56          fwrite(&kcolB,bgroesse,1,Ziel);                /* Block schreiben */
57          NachrichtAnBeagle3(f,"weiter");         /* Bearbeitung fortsetzen */
58        } else {
59          fwrite(&kcolB,sz-(i-1)*bgroesse,1,Ziel);   /* Restblock schreiben */
60          ende = true;                          /* Bearbeitungsende erreicht */
61        }
62      } /* while (!ende) */
63      NachrichtAnBeagle3(f,"Ende");  /* Bearbeitungsende an Beagle 3 melden */
64      fclose(Quelle);
65      fclose(Ziel);
66    }
67  } /* Ende DateiBloeckeVerarbeiten */
```

Mit dem vorstehenden Beispieltext ist ein Muster gegeben, wie man eine Folge von Datenblöcken von den Prozessoren 1 oder 2 im Wechselspiel mit dem Prozessor 3 ver- beziehungsweise entschlüsselt. Die Dateien werden auf dem USB-Stick erwartet, der in `/media/usb-drive` eingehängt wurde (siehe Zeilen 14 und 16). Die Liste der Dateinamen befindet sich in der globalen Liste `liste`, die Länge dieser Liste in `lstlng`, der Index der ausgewählten Datei in der globalen Variablen `dsel` (siehe Zeilen 15 und 17).

Ein mögliches, einfaches Zusammentragen der Dateiliste `liste` sei im Folgenden ebenfalls durch ein Muster angegeben. Es werden lediglich die Dateinamen übertragen, die sich im Hauptverzeichnis des USB-Sticks befinden, der in `/media/usb-drive` hängt.

```
 1  static int filter (const struct dirent *unused) {        /* Filtermethode */
 2    if (strlen(dir->d_name) > 2) /* kurze Namen herausfiltern */
 3      return 1;
 4    else
 5      return 0;
 6  }
 7
 8  void ListeLesen() {
 9    /* Liest die Dateinamen ein, die ver-/entschlüsselt werden könnten.     */
10    /* Die Dateinamen stehen anschließend im globalen Array <liste>, deren  */
11    /* Länge in der globalen Variablen <lstlng>.                            */
12    struct dirent **eps;                        /* Unterverzeichniseintrag */
```

```
13    int n;
14    char* tmp;
15    lstlng = 0;                                    /* Länge der Dateiliste */
16    n = scandir("/media/usb-drive/",&eps,filter,alphasort);
17    if (n >= 0) { /* Unterverzeichnis ist nicht leer */
18      for (int i = 0; i < n; i++) {        /* Dateinamen in Liste eintragen */
19        strcpy(liste[lstlng++],eps[i]->d_name);
20      }
21      free(eps); /* Speicher freigeben */
22    } else {
23      perror ("+++ Unterverzeichnis lässt sich nicht lesen");
24    }
25  } /* Ende ListeLesen */
26
27  void ListeUebertragen(int f) {
28    /* Überträgt die Liste der Dateinamen <liste> an den Prozessor 3.       */
29    char tmp[10];                                          /* Zwischenablage */
30    sprintf(tmp,"%i",lstlng);            /* Übertragung der Länge der Liste */
31    NachrichtAnBeagle3(f,tmp);
32    for (int i = 0; i < lstlng; i++) {        /* Übertragung der Dateinamen */
33      NachrichtAnBeagle3(f,liste[i]);         /* an Beagle 3                */
34    }
35  } /* Ende ListeUebertragen */
```

Mit dem Unterprogramm `filter` in den Zeilen 1 bis 6 werden Dateinamen herausgefiltert, die nicht zur Ver- oder Entschlüsselung angeboten werden sollen. Der `sprintf`-Befehl in Zeile 30 bewirkt die Wandlung der ganzzahligen Größe `lstlng` in einen Text, der an den Prozessor 3 gesendet wird (Zeile 31). Dadurch wird diesem bekannt, wie viele Dateinamen anschließend über k_1 eintreffen werden.

6 Tabellarische Befehlsübersicht

6.1 Matrixbefehle

Der `plumChip` enthält eine binäre Matrix M der Größe $n \times n$. n ist immer ein Vielfaches von 8. Die Matrix M befindet sich in den RAM-Blöcken des FPGA.

Befehl	Beagle	Beschreibung
`C`	3	Der Inhalt der Matrix M des `plumChip`s wird gelöscht, das heißt, an allen $n \cdot n$ Matrixpositionen wird eine Null eingetragen.
`#`	3	Das Frage-Antwort-Tupel (Q, L) wird nach der Langzeitregel (`lerne`) in die Matrix M des `plumChip`s eingetragen.
`!`	3	Das Frage-Antwort-Tupel (Q, L) wird nach der Kurzzeitregel (`merke`) in die Matrix M des `plumChip`s eingetragen.
`?`	3	Der `plumChip` fragt seine Matrix M mit dem Inhalt des Frageregisters Q ab (`beantworte`, `zeige`).
`W`	3	Der `plumChip` fragt seine Matrix M mit dem aktuellen Antworttupel ab (`frageweiter`).
`<A`	3	Der `plumChip` gibt das Antworttupel mit Hilfe der Zeichen '0' und '1' über die Schnittstelle 1,
`>A`	3	über die Schnittstelle 2,
`=A`	3	über die Schnittstelle 3 aus.
`<a`	3	Der `plumChip` gibt das Antworttupel binär über die Schnittstelle 1,
`>a`	3	über die Schnittstelle 2,
`=a`	3	über die Schnittstelle 3 aus.
`+`	3	Der Prüfwert für die Abfrage der Zählerstände wird um 1 heraufgesetzt.
`-`	3	Der Prüfwert für die Abfrage der Zählerstände wird um 1 herabgesetzt.
`p`	3	Der aktuelle Prüfwert wird als vierziffrige Zahl über die Schnittstelle 3 ausgegeben.
`e`	3	Es wird die Anzahl an Einsen im aktuellen Antworttupel als vierziffrige Zahl über die Schnittstelle 3 ausgegeben.
`y`	3	Alle Zeilen der Matrix M werden von einem (Pseudo-) Zufallszahlengenerator genau zur Hälfte mit Einsen gefüllt. Die Saat für den Zufallsgenerator wird über den Befehl `D` eingegeben (siehe *Verwaltungsbefehle*).
`z`	3	Alle Spalten der Matrix M werden von einem (Pseudo-) Zufallszahlengenerator genau zur Hälfte mit Einsen gefüllt. Die Saat für den Zufallsgenerator wird über den Befehl `D` eingegeben (siehe *Verwaltungsbefehle*).
`M`	3	Der Inhalt der Matrix M des `plumChip`s wird über die Schnittstelle 3 mit Hilfe der Zeichen '0' und '1' ausgegeben. Dieser Befehl dient der Kontrolle des Matrixinhalts, zum Beispiel der Prüfung, ob die Hälfte jeder Matrixspalte tatsächlich mit Einsen belegt ist.

6.2 Registerbefehle

Im `plumChip` befinden sich ein Lernantwortregister L und ein Frageregister Q der Länge n und ein weiteres Register R gleicher Länge mit den Zählerständen der letzten Abfrage.

6.2.1 Frageregister Q

Befehl	Beagle	Beschreibung
`,Q[01011]`	3	Der `plumChip` nimmt die Werte (die Zeichen '0' oder '1') eines Fragetupels über Schnittstelle 1 entgegen und legt sie im Frageregister Q ab. Als '1' wird hexadezimal x31 erwartet, alle anderen Zeichen werden als '0' interpretiert.
`.Q[01011]`	3	wie `,Q`, aber über die Schnittstelle 2
`:Q[01011]`	3	wie `,Q`, aber über die Schnittstelle 3
`,q[bits]`	3	Der `plumChip` nimmt die Werte eines Fragetupels binär über Schnittstelle 1 entgegen und legt sie im Frageregister Q ab. Es werden bei der Eingabe $\frac{n}{8}$ Bytes erwartet.
`.q[01011]`	3	wie `,q`, aber über die Schnittstelle 2
`:q[01011]`	3	wie `,q`, aber über die Schnittstelle 3
`<F`	3	Der Inhalt des Frageregisters Q wird mittels der Zeichen '0' und '1' über die Schnittstelle 1 ausgegeben.
`>F`	3	wie `<F`, aber über die Schnittstelle 2
`=F`	3	wie `<F`, aber über die Schnittstelle 3
`<f`	3	Der Inhalt des Frageregisters Q wird binär über die Schnittstelle 1 ausgegeben.
`>f`	3	wie `<f`, aber über die Schnittstelle 2
`=f`	3	wie `<f`, aber über die Schnittstelle 3
`x`	3	Das Frageregister Q wird von einem (Pseudo-) Zufallsgenerator zur Hälfte mit Einsen gefüllt. Die Saat für den Zufallsgenerator wird über den Befehl `D` eingegeben (siehe Verwaltungsbefehle).

6.2.2 Lernantwortregister L

Befehl	Beagle	Beschreibung
`,L[01011]`	3	Der `plumChip` empfängt die Werte eines Lernantworttupels über Schnittstelle 1 und legt sie im Lernantwortregister L ab. Die Anmerkungen zu den Werten '0' und '1' des Kommandos `Q` gelten hier ebenfalls.
`.L[01011]`	3	wie `,L`, über Schnittstelle 2
`:L[01011]`	3	wie `,L`, über Schnittstelle 3
`,l[bits]`	3	Der `plumChip` empfängt die Werte für das Lernantwortregister L binär, byteweise über die Schnittstelle 1 bis damit die n Bits von L gefüllt sind. Es werden bei der Eingabe also $\frac{n}{8}$ Bytes erwartet.
`.l[bits]`	3	wie `,l`, aber über die Schnittstelle 2
`:l[bits]`	3	wie `,l`, aber über die Schnittstelle 3
`<T`	3	Der Inhalt des Lernantwortregisters L wird mittels der Zeichen '0' und '1' über die Schnittstelle 1 ausgegeben.
`>T`	3	wie `<T`, aber über die Schnittstelle 2
`=T`	3	wie `<T`, aber über die Schnittstelle 3
`<t`	3	Der Inhalt des Lernantwortregisters L wird binär über die Schnittstelle 1 ausgegeben.
`>t`	3	wie `<t`, aber über die Schnittstelle 2
`=t`	3	wie `<t`, aber über die Schnittstelle 3

6.3 Schlüsselgenerierung

Befehl	Beagle	Beschreibung
`n`	3	Eine nächste Iterierte wird mit Hilfe des MINA-Verfahrens gebildet und im Frageregister Q abgelegt. Die Iterierten dienen dann dem Verschlüsseln des Inhalts von L mittels des Befehls `o` (XOR). Über die Schnittstelle 3 wird durch Ausgabe einer entsprechenden Anzahl von Zeichen '■' (Zeichen #254) darüber informiert, wie viele Prüfwerte in die Bildung der Iterierten einbezogen wurden.
`N`	3	Die nächste Iterierte wird mit Hilfe des LSB-Verfahrens gebildet und im Frageregister Q abgelegt.
`i`	3	Die nächste Iterierte wird mit Hilfe des Sigma-Verfahrens gebildet und im Frageregister Q abgelegt.
`o`	3	Das Frageregister Q und das Lernantwortregister L werden XOR-verknüpft. Das Ergebnis der XOR-Verknüpfung befindet sich anschließend in L.

6.4 Verwaltungsbefehle

Befehl	Beagle	Beschreibung
`I`	1, 2, 3	Der `plumChip` liefert eine kurze Beschreibung in der Form `imbit.net,C5,SFTP5,v1.8.1,1024`, um damit über FPGA, Experimentierbrett, Version des Befehlssatzes und insbesondere die Größe der `plumChip`-Matrix M Auskunft zu geben.
`S`	1, 2, 3	Im Sekundentakt wird die Meldung `>ok<` über die Schnittstellen 1, 2 und 3 ausgegeben. Der Befehl schaltet die Ausgabe ein und auch wieder aus. Damit soll die korrekte Verbindung von den drei Schnittstellen zu den Prozessoren überprüft werden können.
`'`	3	Die im Sekundentakt gegebene Meldung `>ok<` (siehe Befehl `S`) wird durch die Eingabe des Apostroph `'` ab sofort unterdrückt.
`D`	3	Die Saat für den Zufallsgenerator des `plumChip` wird durch die Eingabe von dreißig Bytes über die Schnittstelle 3 empfangen. Die Saat hat also einen Umfang von 240 Bit.
`d`	3	Der Inhalt des Zufallsgenerators des `plumChip` wird über die Schnittstelle 3 ausgegeben.
`s`	3	Es wird ein Salz der Länge 128 Bit über die Schnittstelle 3 ausgegeben.
`w`	3	Es wird die Eingabe eines Bytes erwartet, dessen Wert die Länge der Meldungen angibt, die die drei Prozessoren untereinander austauschen. Voreingestellt ist die Länge 5.

6.5 Kommunikationsbefehle

Befehl	Beagle	Beschreibung
`1`	3	Es wird eine Meldung von der Schnittstelle 3 zur Schnittstelle 1 übertragen. Die Länge der Meldung wird durch den Befehl `w` eingegeben (siehe Verwaltungsbefehle). Diese Befehle können genutzt werden, um beispielsweise Dateinamen weiterzureichen.
`2`	3	Es wird eine Meldung von der Schnittstelle 3 zur Schnittstelle 2 übertragen.
`3`	3	An Schnittstelle 1 wird die Meldung `>go<` ausgegeben und dann eine Meldung von der Schnittstelle 1 zur Schnittstelle 3 erwartet.
`4`	3	An Schnittstelle 1 wird die Meldung `>go<` ausgegeben und dann eine Meldung von der Schnittstelle 1 zur Schnittstelle 2 erwartet.
`5`	3	An Schnittstelle 2 wird die Meldung `>go<` ausgegeben und dann eine Meldung von der Schnittstelle 2 zur Schnittstelle 3 erwartet.
`6`	3	An Schnittstelle 2 wird die Meldung `>go<` ausgegeben und dann eine Meldung von der Schnittstelle 2 zur Schnittstelle 1 erwartet.

7 Literaturnachweis

[BentzDierks13] Hans-Joachim Bentz/Andreas Dierks: „Neuromathematik und Assoziativmaschinen“, Springer Vieweg, Berlin Heidelberg 2013, ISBN 978-3-642-37937-6

[Schneier15] Bruce Schneier: „Applied Cryptography, Second Edition — Protocols, Algorithms, and Source Code in C — 20th Anniversary Edition“, John Wiley & Sons, Indianapolis 2015, ISBN 978-1-119-09672-6

[Dierks18-1] imbit/Andreas Dierks: „MINA-Verschlüsselungsverfahren“, Teil 1 (2. September 2018), imbit, Hildesheim 2018

[Dierks18-7] imbit/Andreas Dierks: „MINA-Verschlüsselungsverfahren MINA, LSB, Sigma“, Teil 7 (4. Dezember 2018), imbit, Hildesheim 2018

[Dierks19-7] imbit/Andreas Dierks: „LFSR für plumChip v2.0“, 30. Oktober 2019, imbit, Hildesheim 2019