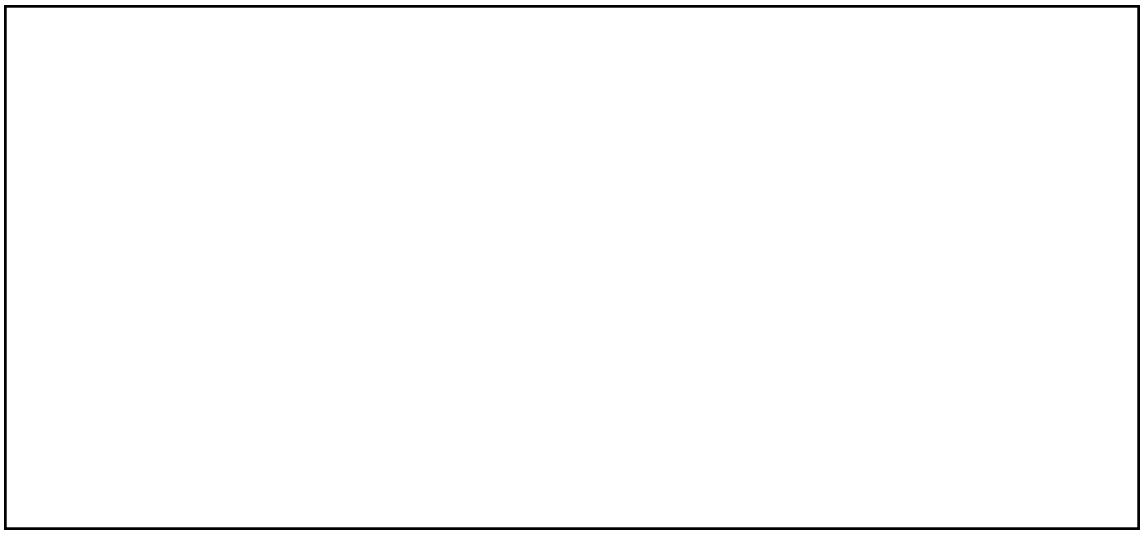
Fast Diagnose

Applikation zur Schnelldiagnose von ABB-Robotern

Von: Manuel-Leonhard Rixen

Bearbeitungszeitraum: 08/15 - XX/XX



Kurzfassung

Um in möglichst kurzer Zeit wichtige Informationen über ein System mit integrierten Industrierobotern zu erhalten dient die *Fast Diagnose*. Die auszugebenden Informationen beinhalten die Taktzeit (Aktual- und Durchschnittswert) mit grafischer Darstellung, sowie Konsolenausgabe und Anlagenparameter. Zudem wird der Benutzer anhand von Events über einen Zyklusstopp informiert. Aktuell werden ausschlielich Roboter des Herstellers ABB unterstützt.

Inhalt

1 Einleitung

Allgemeine Einleitung. Erläuterung zum Ausgangspunkt. Erläuterung zum Kapitelinhalt.

2 Funktion und Aufbau

2.1 Nachrichtenaufbau

Jede Nachricht, die von der ABB-Steuerung gesendet und von der Application (App) korrekt empfangen werden soll, muss folgendes Format aufweisen:

Nachricht = :{command}:{message};

Bei der Standardkommunikation sind für {command} folgende Parameter bereits fest vergeben:

{0...n-1}: Machine data mit n aus $\mathbb{N}$
Dieser Parameter kann erweitert werden. Hierbei muss die App (Android), als auch das Modul (ABB) angeasst werden.

{l}: Logging

{c1}: Cycle time (Aktueller Wert)
{c2}: Cycle time (Durchschnittswert)
Diese Parameter können erweitert werden mit z.B. {c3}, {c4}, etc.

{e}: Events

{p}: Ping
Dieser Parameter wird zur Überprüfung verwendet, ob der Empfänger noch vorhanden ist. Dies wird in 2.2.1 erläutert.

Sofern in einer der Nachrichtenkomponenten ({command} oder {message}) ein Doppelpunkt enthalten ist, so muss dieses mit einem führenden Doppelpunkt angegeben werden. Ansonsten wird es nicht als, im Satz enthaltenes, Zeichen erkannt. Die Nachricht wird dadurch fehlerhaft dargestellt.

Nachrichtenbeispiele bei der Standardkommunikation sind folgend aufgeführt:

Machine data
:0:Multiflex;
:1:20184;
:2:2015;
:3:10.10.15;

Tabelle 2.1: Parameter der Maschinendaten

Maschinentyp	Maxiflex
Projektnummer	20184
Herstelljahr	2015
Werksabnahme	10.10.15
Produktionsstart	15.10.15
Serien-Nr.	20184
Software-Version	20184
Robotertyp	20184
Steuerungs-ID	20184
IP-Adresse	192.168.1.2
Sprache	DE
Betriebsstunden	10.5

Die Maschinendaten beinhalten die wichtigsten Informationen über das System und werden als {command}, beginnend bei 0, durchnummeriert.
Aktuell werden die in 2.1 dargestellten Parameter ausgegeben. Die in der rechten Spalte aufgezeigten Werte sind als Beispiel anzunehmen.

Logging
:l:Gesamtzahl der gefertigten Teile 81;

Cycle time
:c1:2.545;
:c2:2.273;

Events
:e:1::1::10::1::X::X::X::X;
:e:1::1::17::1::T_ROB1::X::X::X;
:e:3::28::10::1::63::7::X::X;

2.2 Programmaufbau

2.2.1 ABB

Das System besteht aus mehreren Tasks von denen vier für die Kommunikation zwischen dem Abb-Controller und der Android-App fungieren. Die jeweilige Kernaufgabe der Tasks werden im Folgendenerläutert und die wichtigsten Code-Abschnitte aufgezeigt.

2.2.1.1 Task ServerComm

Der Task *ServerComm* beinhaltet das Modul *ServerComm* und ist für die grundsätzliche Kommunikation, d.h. für das Senden und Empfangen von Daten, zuständig. In den Zeilen 1 - 5 des Listings 2.2.1.1 ist die Hauptroutine zu sehen, in welcher der Routinen-Aufruf *waitForClients()* endlos erfolgt. In dieser Routine wird der Socket bis von Case 1 bis einschließlich Case 3 (Zeile 9 - 10) initialisiert und in Case 4 (Zeile 14) auf eine Client-Verbindung gewartet. Sobald ein Client verbunden ist durchläuft die For-Schleife in Zeile 18 - 26 insgesamt acht Buffer (jeder Task besitzt zwei Buffer). Während ein Buffer die zu sendenen Daten enthält, ist im zweiten Buffer ein Zustand hinterlegt, ob Daten (für den jeweiligen Buffer) vorhanden sind.

Sofern ein Event zum Senden bereit liegt, der Buffer also ein Element an der Position i enthält, ist der Wert von bufferStateEvent{i} in Zeile 19 wahr.
In Zeile 20 wird das Datenpaket (String) zum Client gesendet. Zeile 21 - 23 zeigen das Zurücksetzen des Buffers und eine Wartezeit, welche notwendig ist, da sonst die Sendefrequenz zu hoch ist (Datensalat).

Falls die Verbindung zusammenbricht oder ein ähnlicher Fehler erfolgt, der die Kommunikation behindert, wird in Zeile 31 - 38 die Fehlerbehandlungsroutine aufgerufen. Darin erfolgt der Aufruf von *initSocket()*, wodurch die Verbindung beendet und neu initialisiert wird. Der Programmzeiger ist anschließend innerhalb kürzester Zeit (max. ca. 1s) wieder in Zeile 14 (warten auf eingehende Verbindung).
Um die Verbindung dauerhaft zu prüfen muss immer gesendet werden, da so der Aufruf von *SocketSend* einen Fehler generiert.

Listing 2.1: Task ServerComm - Modul ServerComm

```
1 PROC main()
2     WHILE TRUE DO
3         waitForClients;
```

```
4       ENDWHILE
5   ENDPROC
6   (...)
7   PROC waitForClients()
8        TEST state
9        CASE 1:
10        (...)
11       CASE 4:
12           WHILE listening DO
13               IF i<=MAX_CLIENTS THEN
14                   SocketAccept server_socket,client_socket{i}\Time:=WAIT_MAX;
15            (...)
16            ENDIF
17           ENDWHILE
18          FOR i FROM 1 TO 25 DO
19            IF (bufferStateEvent{i}) THEN
20              SocketSend client_socket{1}\Str:=sendbufferEvent{i};
21              sendbufferEvent{i} := "";
22              bufferStateEvent{i} := false;
23              WaitTime 0.08;
24            ENDIF
25            (...)
26          ENDFOR
27       DEFAULT:
28       ENDTEST
29  ENDPROC

31      ERROR
32          IF ERRNO=ERR_SOCK_CLOSED THEN
33      (...)
34          IF ERRNO=ERR_SOCK_ADDR_INUSE THEN
35      (...)
36          IF ERRNO=ERR_SOCK_TIMEOUT THEN
37      (...)
38      ENDPROC

40      PROC initSocket()
41          SocketClose server_socket;
42          SocketClose client_socket{1};
43      (...)
44          listening:=TRUE;
45      clientConnected:=FALSE;
46      ENDPROC
```

2.2.1.2 Task EventMessages

Der Task *EventMessages* beinhaltet das Modul *EventMessages* und ist dafür zuständ alle Informationen, Warnungen und Fehler zu senden.
Beispiel: Wenn der Roboter im Automatikbetrieb ein Programm abarbeitet und das Programm gestoppt wird erfolgt das Senden einer entsprechenden Dialog-Nachricht (vom Inhalt her wie auf dem Flexpendant). Das Modul besteht im Grunde nur aus ei-

nem Interrupt *err_trap*, dessen Aufruf erfolgt sobald eine Ereignismeldung ansteht. Um den Interrupt aktiv zu halten wird in der Hauptroutine, nach der Definition des Interrupts, eine Endlos-Schleife durchlaufen (s. 2.2.1.2 Zeile 1 - 7). Im Interrupt selbst erfolgt das Setzen eines Flags mit dem das Detektierten eines Zyklusstop im Automatikbetrieb erfolgt. In Zeile 18 - 19 wird der zu sendende String an die Routine *tpWriteSocket()* übergeben.
Die Routine *tpWriteSocket()* schreibt u.A. den übergebenen String in den entsprechenden Buffer des Tasks, sobald ein Client verbunden ist (s. Zeile 23 - 30). Der String setzte sich aus folgenden Komponenten zusammen:
msgType+robotState+::+err_domain+::+err_number+::+err_type+::+str1 wodurch folgender Beispielstring vorliegen kann:
:e:1::1::10::1::7

Bedeutung der Parameter:
msgType: Nachrichtentyp *e* für Event
robotState: Status des Zyklus (Motoren an/aus)
err_domain: Fehlerdomäne (Auswahl der entsprechenden xml-Datei)
err_number: Fehlernummer (Auswahl des entsprechenden Stings aus der xml-Datei)
err_type: Fehlertyp (Info, Warnung, Fehler)
str1: Erster zusätzlicher Textparameter

Der String im Code enthält mehrere X-Zeichen. Diese fungieren als Platzhalter, da ansonsten die Gesamtlänge des Strings (max. 80 Characters) zu hoch wäre, würden alle Parameter (str1 - str5) eingebettet sein.

Eine Beschreibung der einzelnen Parameter kann in der entsprechenden ABB-Dokumentation nachgeschlagen werden.

Listing 2.2: Task EventMessages - Modul EventMessages

```
1  PROC main()
2      CONNECT err_int WITH err_trap;
3      IError COMMON_ERR,TYPE_ALL,err_int;
4      WHILE TRUE DO
5        WaitTime 0.01;
6      ENDWHILE
7  ENDPROC

9  TRAP err_trap
10     IF (DOutput(DOF_CycleOn))=1 AND firstCycleStart=0 THEN
11         robotState:=1;
12         firstCycleStart:=1;
13     ENDIF
14     IF (DOutput(DOF_CycleOn)=0) AND (firstCycleStart=1) THEN
15         robotState:=0;
```

```
16          firstCycleStart:=0;
17      ENDIF
18      tpWriteSocket
19      ValToStr(robotState)+"::"+ValToStr(err_domain)+"::"+ValToStr(err_number)+"::"+
            ValToStr(err_type)+"::"+str1+"::"+"X"+"::"+"X"+"::"+"X"+"::"+"X",":e:";
20  ENDTRAP

22  PROC tpWriteSocket(string msg,string msgType)
23    IF clientConnected THEN
24      sendbufferEvent{cntr}:=msgType+msg+";";
25      bufferStateEvent{cntr}:=TRUE;
26      cntr:=cntr+1;
27      IF (cntr>=25) THEN
28        cntr:=1;
29      ENDIF
30    ENDIF
31  ENDPROC
```

2.2.1.3 Task MachineData

Der Task *MachineData* beinhaltet das Modul *MachineData* und ist dafür zuständig die Maschinendaten zu senden. Diese Daten beinhalten in der aktuellen Version die in 2.1 dargestellten Parameter. Sobald ein Client verbunden ist, beginnt das Senden eines String-Arrays mit der Routine *tpWriteSocket()* innerhalb einer For-Schleife (s. 2.2.1.3 Zeile 5 - 12). In der Routine *tpWriteSocket()* ändert sich nur der zu beschreibende Buffer entsprechend des Tasks.

Listing 2.3: Task MachineData - Modul MachineData

```
1   PROC main()
2     (...)
3     isSending := TRUE;
4     mData:=[machineName,projectNumber,dateOfCreation,dateOfFAT,dateOfSOP,serial,version
          ,rtype,cid,lanip,clang,dutyTime];
5       WHILE isSending DO
6         IF (clientConnected AND isSending) THEN
7             FOR i FROM 1 TO MAX_DATA_SIZE DO
8                 tpWriteSocket mData{i},":"+ValToStr(i-1)+":";
9               ENDFOR
10        isSending:=FALSE;
11          ENDIF
12      ENDWHILE
13    ! Give other threads enough time to process
14    WaitTime 5;
15  ENDPROC

17  PROC tpWriteSocket(string msg,string msgType)
18    IF clientConnected THEN
19      sendbufferMdata{cntr}:=msgType+msg+";";
20      bufferStateMdata{cntr}:=TRUE;
21      cntr:=cntr+1;
```

```
22      IF (cntr>=25) THEN
23        cntr:=1;
24          ENDIF
25    ENDIF
26  ENDPROC
```

2.2.1.4 Task CycleTimer

Der Task *CycleTimer* beinhaltet das Modul *CycleTimer* und wurde deshalb in einen externen Task implementiert, da sonst die Taktzeitaufzeichnung den Bewegungstask zu stark belasten (Latenzen).
Das Modul besteht vom Prinzip her aus den Interrupts *nextCTtrap* und *resetCTtrap* (s. 2.2.1.4 Zeile 8 - 14). Um diese aktiv zu halten wird in der Hauptroutine eine Endlos-Schleife durchlaufen (s. Zeile 1 - 6). Durch den Aufruf des Interrupts *nextCTtrap* erfolgt der Routinen-Aufruf von *NextCycleTime()*, wodurch die entsprechenden Daten an den Socket gesendet werden (s. Zeile 18 - 21). In Zeile 18 - 19 wird der Routine *tpWriteSocket()* als msgType :l: übergeben. Hierdurch wird der übergebene String geloggt. Zeile 20 und 21 sendet die aktuelle Zykluszeit und dessen Durchschhnitt (msgType :c1: und :c2:).

Listing 2.4: Task CycleTimer - Modul CycleTimer

```
1   PROC main()
2     (...)
3     WHILE TRUE DO
4       WaitTime 0.01;
5     ENDWHILE
6   ENDPROC

8   TRAP nextCTtrap
9     NextCycleTime;
10  ENDTRAP

12  TRAP resetCTtrap
13    ResetCycleTimes;
14  ENDTRAP

16  PROC NextCycleTime()
17    (...)
18      tpWriteSocket "Gesamtzahl der gefertigten Teile::
19      "+ValToStr(nCyclesShow),":l:";
20    tpWriteSocket ValToStr(nCycleTime),":c1:";
21    tpWriteSocket ValToStr(cycleTimeMean{1}),":c2:";
22    (...)
23  ENDPROC

25  (...)

27  PROC tpWriteSocket(string msg,string msgType)
```

```
28    IF clientConnected THEN
29      sendbufferCycleTime{cntr}:=msgType+msg+";";
30      bufferStateCycleTime{cntr}:=TRUE;
31      cntr:=cntr+1;
32      IF (cntr>=25) THEN
33        cntr:=1;
34          ENDIF
35    ENDIF
36  ENDPROC
```

Die Buffergröße beläuft sich aktuell auf 25 Einträge.

2.2.2 Android

Das System besteht aus mehreren Klassen von denen die wichtigsten folgend Erläuterung finden. Die Klassen sind der Reihenfolge nach den Tasks der Robotersteuerung zuzuordnen. D.h. die Klasse *Receiver* ist für die grundsätzliche Kommunikation zuständig - entsprechend des Tasks *ServerComm*, etc.

2.2.2.1 Klasse Receiver

Die Klasse *Receiver* ist für das Senden und Empfangen von Daten zuständig. Die grundsätzliche Kommunikation erfolgt in der Klasse NetClient, welche innerhalb des Konstruktors (s. 2.2.2.1 Zeile 1 - 5) als Objekt erzeugt wird. Die Übergabeparameter bestehen aus der IP und dem Port.

Die Klasse Receiver implementiert ein Runnable, sodass innerhalb der Routine run() die Kommunikation mit einem Client abläuft. In Zeile 9 erfolgt der Verbindungsaufbau. Ist dieser erfolgreich, so beginnt das Empfangen von Daten innerhalb einer While-Schleife (Zeile 11 - 20). Die empfangenen Daten werden durch Events an alle Teilnehmer (Klasse Events, MachineData und CycleTime) gesendet innerhalb derer die Nachricht entsprechend des Inhalts von msgType extrahiert wird.
Ist die Verbindung nicht erfolgreich, so erfolgt die Ausgabe einer Meldung auf dem Display und die App wird nach einer Zeit x beendet (s. Zeile 22 - 32).

Mit der Routine *registerListener()* erfolgt das Registrieren auf den Event-Generator. Dieses Interface beinhaltet die Methode onEvent(), die zwei Übergabeparameter aufweist, um den Nachrichtentyp (msgType z.B. :e: für Event) und die Nachricht selbst zu übertragen.

Listing 2.5: Klasse Receiver

```
1     public Receiver(Context context, String ip, String port, Activity activity){
2         (...)
3         if (nc == null) nc = new NetClient(this.ip,
4         Integer.parseInt(this.port));
5     }
6   (...)

8  public void run() {
9    if (nc.connectWithServer()) {
10       (...)
11         while (isRunning) {
12           data = nc.receiveDataFromServer();
13       if ((data[0] != null) && (data[1] != null)) {
14                (...)
15                   for (EventListener eventListener : listeners)
16                     eventListener.onEvent(data[0], data[1]);
17                       (...)
18             }
19         }
20     }
21   nc.disConnectWithServer();
22     else {
23     (...)
24         Toast.makeText(context, R.string.connectionError,
25         Toast.LENGTH_LONG).show();
26         Timer t = new Timer();
27         t.schedule(new TimerTask() {
28         public void run() {
29         activity.finish();
30     }
31   }, maxActivityShowTime);
32 }

34 public void registerListener(EventListener listener) {
35   this.listeners.add(listener);
36 }

38 public interface EventListener {
39     void onEvent(String data1, String data2);
40   (...)
41 }
```

2.2.2.2 Klasse Events

Der Klasse *Events* definiert einen Tab und empfängt/verarbeitet die von der Robotersteuerung gesendeten Events (Information, Warnung, Fehler). Die Klasse registriert sich (wie in allen anderen Tab-Klassen auch) in ihrer Methode *onCreate()* auf den Event-Generator, bzw. das Interface *EventListener* (s. 2.2.2.2 Zeile 7 - 8).
Sobald die Robotersteuerung einen String sendet, erfolgt der Aufruf von *onEvent()* und einhergehend der Aufruf von *showEvent()* (s. Zeile 12 - 14 und 16 - 22). Daraufhin wird geprüft, ob der Nachrichtentyp (msgType) das Zeichen *e* beinhaltet (s. Zeile

17). Ist dies der Fall so wird der String aufgeteilt und dem Objekt *XMLParsing()* übergeben (s. Zeile 20).

Die Klasse *XMLParsing* erzeugt das String-Array *eventDescription*

, das den gesamten Meldungstext beinhaltet. D.h. anhand der Parameter Fehlerdomäne und Fehlernummer erfolgt die Auswahl der xml-Datei (Zeile 30 - 40 verkürzt dargestellt) in welcher der Beschreibungstext für die Meldung hinterlegt ist.
Das Array *eventMessages*

enthält an der Position 1 die Fehlerdomäne, an Position 2 die Fehlernummer und an Position 3 den Fehlertyp. Alle weiteren Positionen beinhalten die Textparameter.

Das Parsen der entsprechenden xml-Datei erfolgt in Zeile 45 - 78 mit einer Switch-Case Struktur. In dem ersten Case (Zeile 48 - 56) wird (innerhalb von if-Abfragen) der Ort eines xml-Eintrages gesucht und die hinterlegten Strings in das Array *eventDescription*[] kopiert (die Cases sind verkürzt dargestellt). Im zweiten Case (Zeile 70 - 72) wird das Ende des Dokuments detektiert.
Ist das Parsen beendet erfolgt der Methoden-Aufruf *onPostExecute()* (Zeile 80 - 85) und der Meldungstext wird sofort als Dialog mit Vibration dargestellt (sofern die Motoren ausgeschaltet sind, bzw. eventMessages[0] == 0 ist) oder in eine Liste geschrieben.

Die xml-Dateien wurden von der Robotersteuerung extrahiert und in die App-Struktur eingefügt.

Listing 2.6: Klasse Events

```
1  protected void onCreate(Bundle savedInstanceState) {
2  (...)
3    Bundle bundle = getIntent().getExtras();
4      BaseData baseData = (BaseData) bundle.getSerializable("baseData");
5      if (receiver == null) {
6        (...)
7          receiver = baseData.getReceiver();
8          receiver.registerListener(this);
9          (...)
10 }

12 public void onEvent(String msg, String msgType) {
13   showEvent(msgType, msg, true);
14 }

16 private void showEvent(String msgType, String eventMessage, boolean saveEvent) {
```

```
17    if (msgType.equals("e")){
18        if (saveEvent) listViewEntryData[listCounter] = eventMessage;
19          String[] tempMessage = eventMessage.split(":");
20          new XMLParsing(saveEvent, this).execute(tempMessage);
21      }
22  }

24      private class XMLParsing extends AsyncTask<String, Void, String[]> {
25  (...)
26          protected String[] doInBackground(String... eventMessages) {
27              this.eventMessages = eventMessages;
28              String[] eventDescription = new String[] {"", "", "", "", "", ""};
29  (...)
30                  switch (Integer.parseInt(eventMessages[1])){
31                      case 1:
32                          if (Integer.parseInt(eventMessages[2]) <= 175) filename
33                          = "elog/"+"opr_"+"elogtext_"+1+".xml";
34                          if ((Integer.parseInt(eventMessages[2]) > 175) &&
35                          (Integer.parseInt(eventMessages[2]) <= 1231)) filename
36                          = "elog/"+"opr_"+"elogtext_"+2+".xml";
37                          (...)
38                          break;
39                      case 2:
40                          (...)
41                  }
42                  InputStream in_s = getAssets().open(filename);
43                  (...)

45                  while ((event != XmlPullParser.END_DOCUMENT) && !messageReadOk)
46                  {
47                      switch (event){
48                          case XmlPullParser.START_TAG:
49                              name = xmlParser.getName();
50                              if(name.equals("Message") && !messageEntryFound) {
51                                  String messageNumber =
52                                  xmlParser.getAttributeValue(null,"number");
53                                  if (messageNumber.equals(eventMessages[2])){
54                                      messageEntryFound = true;
55                                  }
56                              }
57                              if(messageEntryFound && name.equals("Title"))
58                              eventDescription[0] = xmlParser.nextText();
59                              if(messageEntryFound && name.equals("Description")){
60                                  eventDescription[1] = xmlParser.nextText();
61                                  eventDescription[1] =
62                                  String.format(eventDescription[1],
63                                  eventMessages[4], eventMessages[5],
64                                  eventMessages[6], eventMessages[7],
65                                  eventMessages[8]);
66                              }
67  (...)
68                              break;

70                          case XmlPullParser.END_TAG:
71  (...)
72                      }
73                      if (!messageReadOk){
```

```
74  (...)
75          }
76          eventDescription[5] = eventMessages[3];
77                      return eventDescription;
78                      }

80          protected void onPostExecute(String[] result) {
81              if (addEvents) setEventList(result[0]);
82              if ((eventMessages[0].equals("0")) || !addEvents){
83                  customEventDialog.showDialog(result);
84                  if (addEvents) vibrator.vibrate(800);
85              }
86          }
87      }
```

2.2.2.3 Klasse MachineData

Die Klasse *MachineData* definiert einen Tab und zeigt tabbelarisch die Maschinendaten auf. Die Klasse registriert sich (wie in allen anderen Tab-Klassen auch) in ihrer Methode *onCreate()* auf den Event-Generator, bzw. das Interface *EventListener* (s. 2.2.2.3 Zeile 7 - 8).
Sobald die Robotersteuerung einen String sendet, erfolgt der Aufruf von *onEvent()* und einhergehend der Aufruf von *showEvent()* (s. Zeile 12 - 14 und 16 - 27). Daraufhin wird geprüft, ob der Nachrichtentyp (msgType) eine Zahl (0, 1, 2, n) enthält (s. Zeile 18 - 21). Ist dies der Fall so wird der String der entsprechenden Zeile in der Tabelle übergeben (s. Zeile 24 - 25).

Listing 2.7: Klasse MachineData

```
1       protected void onCreate(Bundle savedInstanceState) {
2   (...)
3           Bundle bundle = getIntent().getExtras();
4           BaseData baseData = (BaseData) bundle.getSerializable("baseData");
5           if (receiver == null) {
6               (...)
7                   receiver = baseData.getReceiver();
8                   receiver.registerListener(this);
9               (...)
10      }

12          public void onEvent(String data1, String data2) {
13              showEvent(data2, data1);
14          }

16          private void showEvent(String msgType, String eventMessage) {
17              int number = -1;
18                      try {
19                          number = Integer.parseInt(msgType);
20                      } catch (NumberFormatException e) {
21                      }
```

```
22              if (number != -1) {
23              (...)
24                  machineData[Integer.parseInt(msgType)] = eventMessage;
25                  textViews[Integer.parseInt(msgType)].setText(eventMessage);
26              }
27              (...)
28          }
```

2.2.2.4 Klasse CycleTime

Die Klasse *CycleTime* definiert einen Tab und stellt den aktuellen, sowie den Durchschnitts-Wert der Zykluszeit dar. Die Klasse registriert sich (wie in allen anderen Tab-Klassen auch) in ihrer Methode *onCreate()* auf den Event-Generator, bzw. das Interface *EventListener* (s. 2.2.2.4 Zeile 7 - 8).
Sobald die Robotersteuerung einen String sendet, erfolgt der Aufruf von *onEvent()* und einhergehend der Aufruf von *showEvent()* (s. Zeile 12 - 14 und 16 - 25). Daraufhin wird geprüft, ob der Nachrichtentyp (msgType) das Zeichen *c1* oder *c2* enthält (s. Zeile 17 - 24). Ist dies der Fall so wird der String dem entsprechenden Textfeld übergeben (s. Zeile 19 und 23).

Listing 2.8: Klasse CycleTime

```
1     protected void onCreate(Bundle savedInstanceState) {
2  (...)
3          Bundle bundle = getIntent().getExtras();
4          BaseData baseData = (BaseData) bundle.getSerializable("baseData");
5          if (receiver == null) {
6              (...)
7                  receiver = baseData.getReceiver();
8                  receiver.registerListener(this);
9              (...)
10     }

12         public void onEvent(String msg, String msgType) {
13             showMessage(msgType, msg);
14         }

16             private void showMessage(String msgType, String msg) {
17                 if (msgType.equals("c1")) {
18                     String msgTemp = msg.replace(".", ",");
19                     cycleTimeViewer_actual.setText(msgTemp);
20                 }
21                 if (msgType.equals("c2")) {
22                     String msgTemp = msg.replace(".", ",");
23                     cycleTimeViewer_mean.setText(msgTemp);
24                 }
25             }
```

2.2.2.5 Klasse Logging

Der Klasse *Logging* definiert einen Tab und stellt die Logging-Nachrichten dar, die der User ansonsten auf dem FlexPendant unt Logging sehen würde. Die Klasse registriert sich (wie in allen anderen Tab-Klassen auch) in ihrer Methode *onCreate()* auf den Event-Generator, bzw. das Interface *EventListener* (s. 2.2.2.5 Zeile 7 - 8). Sobald die Robotersteuerung einen String sendet, erfolgt der Aufruf von *onEvent()* und einhergehend der Aufruf von *showEvent()* (s. Zeile 12 - 14 und 16 - 28). Daraufhin wird geprüft, ob der Nachrichtentyp (msgType) das Zeichen *l* enthält (s. Zeile 17). Ist dies der Fall so wird der String einer Liste hinzugefügt (s. Zeile 22 - 23) mit dem aktuellsten Eintrag an oberster Stelle. Die maximale Länge der Liste beläuft sich aktuell auf 10. Ist diese überschritten, so wird der letzte Listeneintrag gelöscht.

Listing 2.9: Klasse Logging

```
1     protected void onCreate(Bundle savedInstanceState) {
2 (...)
3         Bundle bundle = getIntent().getExtras();
4         BaseData baseData = (BaseData) bundle.getSerializable("baseData");
5         if (receiver == null) {
6             (...)
7                 receiver = baseData.getReceiver();
8                 receiver.registerListener(this);
9             (...)
10    }

12        public void onEvent(String msg, String msgType) {
13            showMessage(msgType, msg);
14        }

16    private void showMessage(String msgType, String msg) {
17        if (msgType.equals("l")) {
18            int MAX_LOG_AMOUNT = 10;
19            if (loggingList.size() >= MAX_LOG_AMOUNT) {
20                loggingList.remove(loggingList.size() - 1);
21            }
22            loggingList.add(0, msg);
23            arrayAdapter.notifyDataSetChanged();
24            int MAX_LOG_COUNTER = 100;
25            if (logCounter <= MAX_LOG_COUNTER - 1) logCounter += 1;
26            else logCounter = 0;
27        }
28    }
```

Alle weiteren Klassen, die bei dem Zusammenspiel zwischen der Robotersteuerung und der App beteiligt sind finden im Folgenden Erläuterung. Dabei wird kurz die Funktion erläutert.

MainActivity
In dieser Klasse werden die Tabs generiert und dargestellt, sowie der Barcode-Reader gestartet und auf dessen Ergbenis gewartet. Ist dieses i.O. so erfolgt das Starten des Receiver-Threads.

BaseData
Diese Klasse ist für die Datenablage gedacht, die von verschiedenen Klassen benötigt werden.

NetClient
Die Klasse *NetClient* ist für die grundsätzliche Kommunikation zuständig. D.h. Verbindungsauf- und abbau, sowie Senden und Empfangen von Daten. In der Methode *receiveDataFromServer()* wird bspw. jedes Character eingelesen.

CustomDialogEvent
Diese Klasse erzeugt den Dialog, welcher alle Komponenten für die Darstellung des Meldungstextes enthält (Information, Warnung, Fehler).

Es existieren noch weitere Dialog-Klassen, welche für bestimmte Anwendungsfälle konzipiert sind. So wird bspw. mit der Klasse *CustomDialogAbout* die About-Information dargestellt. Alle weiteren Dialog-Klassen werden aktuell nicht genutzt.

BarCodeReading
Diese Klasse erzeugt den QR-Code Reader, validiert den Inhalt des Codes und gibt eine Rückmeldung an die Klasse *MainActivity* zurück.

CycleTimeDrawThread
Um die Zykluszeit grafisch darstellen zu können ist diese Klasse hilfreich. Darin enthalten sind Methoden, mit denen einen grafische Ausgabe (Diagramm) erzeugt werden kann.
Aktuell wird die Klasse nicht genutzt.

2.3 Grafische Benutzeroberfläche

2.3.1 Screen

Beim Starten der App erscheint (sofern eine WLan-Verbindung aktiv ist) die in 2.1 dargestellte Benutzerschnittstelle.
Ist der QR-Code korrekt, so kommt die HMI der App zum Vorschein. Hierbei startet als erstes immer der Tab, in dem die Maschinendaten dargestellt werden (s. 2.2).

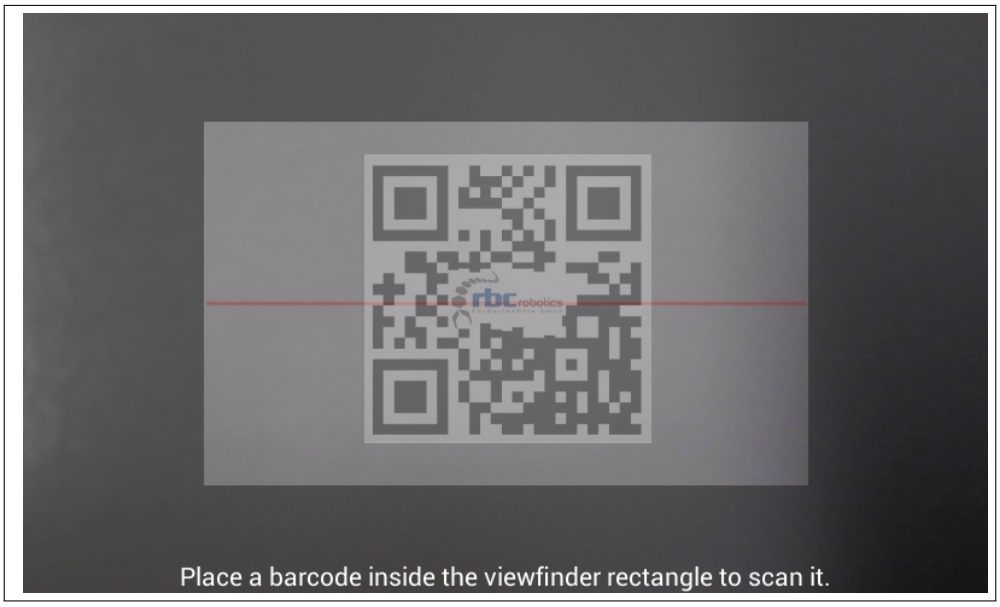


Abb. 2.1: Startscreen

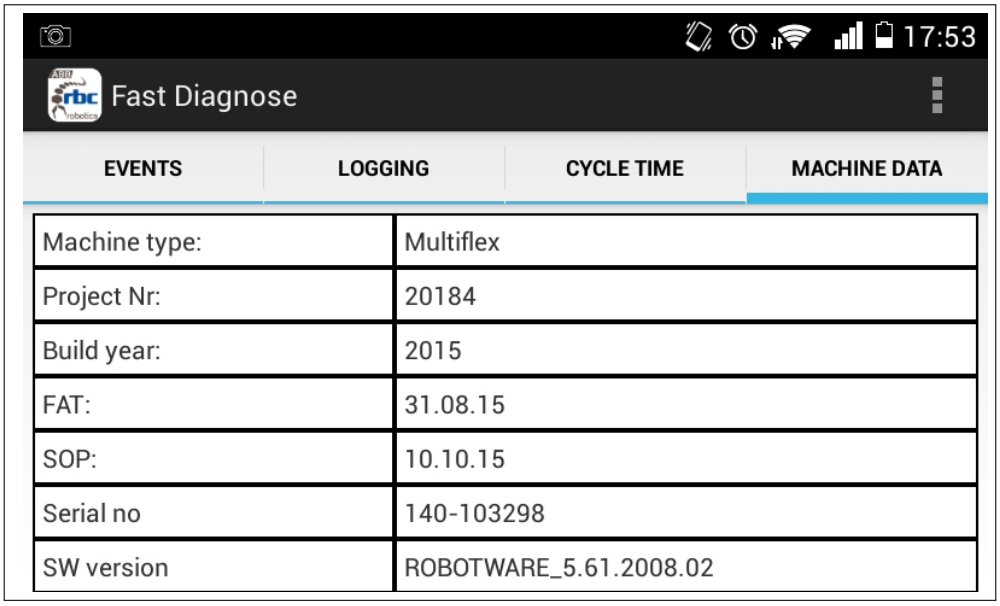


Abb. 2.2: Tab Machine Data

Das Erscheinungsbild aller weiteren Tabs ist in 2.3, 2.4 und 2.5 dargestellt.

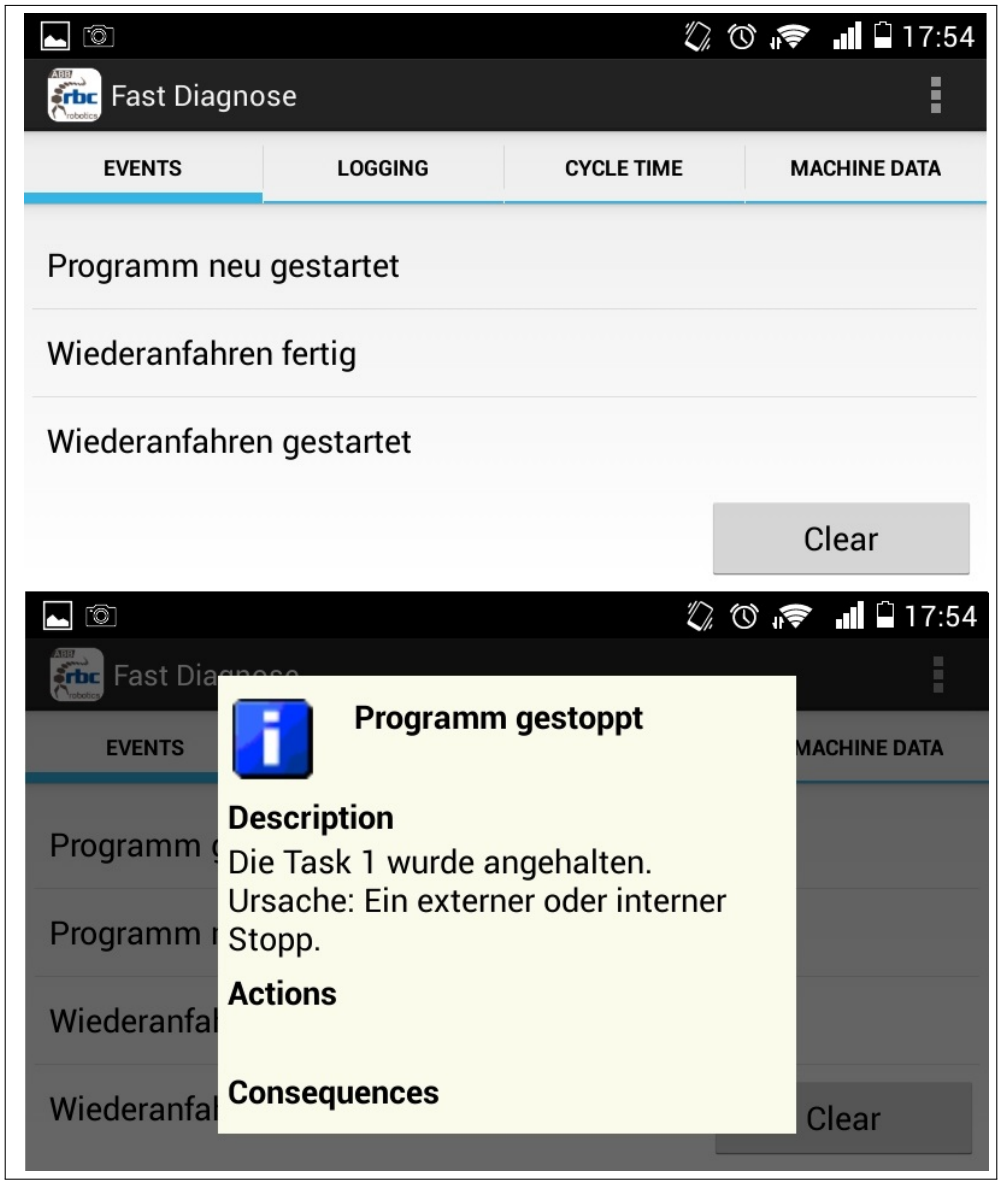


Abb. 2.3: Tab Events (o: Event-Liste, u: Event-Dialog)

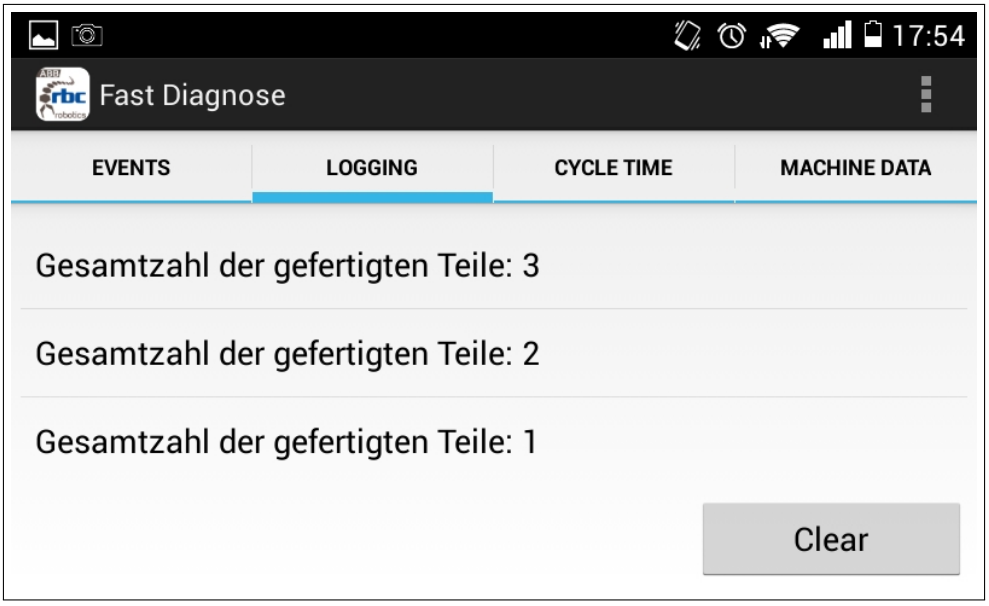


Abb. 2.4: Tab Logging

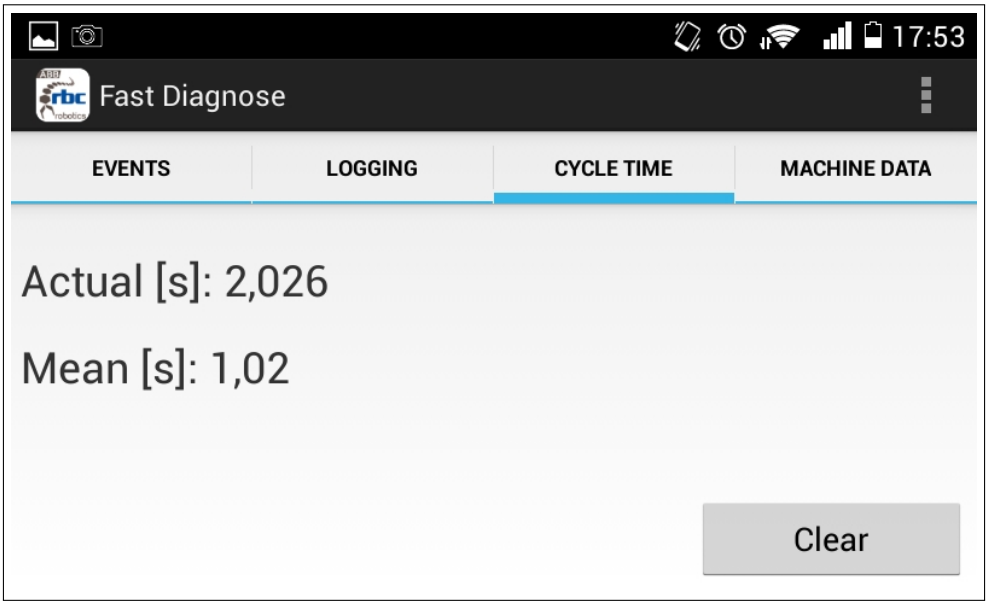


Abb. 2.5: Tab Cycle Time

2.3.2 Icons

Das Launcher-Icon der App ist in 2.6 dargestellt. Hierbei ist das oben links eingefügte Schriftzug zu beachten, welcher die kompatible Robotersteuerung beschreibt (in diesem Fall ABB).

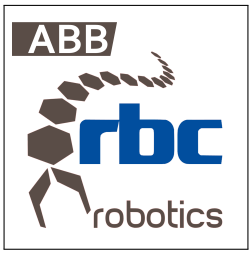


Abb. 2.6: App Icon

Weitere Icons, die in dieser App verwendet werden, sind für den Kontext des Meldungstextes (Event-Nachrichten) vorgesehen (s. 2.7, 2.8 und 2.9)

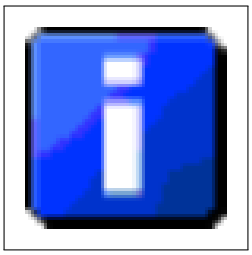


Abb. 2.7: Icon Meldungstext Information

Abb. 2.8: Icon Meldungstext Warnung

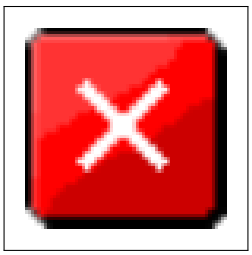


Abb. 2.9: Icon Meldungstext Fehler

3 Ausblick

Die App ist in der aktuellen Version noch stark ausbaufähig. Der wichtigste Aspekt betrifft das Hinzufügen der Kompatibilität für eine Kuka-Steuerung und eventuell weitere Robotersteuerungen.
Eine vollständige Lister der Verbesserungen sind folgend dargestellt.

Offene Punkte:

- Kompatibilität für Kuka-Steuerung hinzufügen
- Kommunikation in Background (Service) verlagern
- Speichern und Auswählen bisheriger Verbindungen
- Vollständige Meldungsparameter (str1, str2, etc.) übertragen
- Verbindung von mehreren Clients ermöglichen
- Diagramme für Zykluszeit hinzufügen
- Handshake beim Senden von Daten implementieren

Listings

Abbildungsverzeichnis

Tabellenverzeichnis