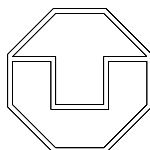
Technische Universität Dresden

Fakultät Elektrotechnik und Informationstechnik

Fraunhofer Institut

Institut für mikroelektronische Schaltungen und Systeme



Diplomarbeit

zum Erlangen des akademischen Grades Diplomingenieur (Dipl.-Ing.)

Thema: Untersuchung und Entwicklung von Konzepten für eigensichere Sensorsysteme

Vorgelegt von: Ralf Hildebrandt (Ralf-Hildebrandt@gmx.de)
(22.03.1978, Bautzen)

Betreuer: Dipl.-Ing. Peter König

Verantwortlicher Hochschullehrer: Prof. Dr.-Ing. habil. W. J. Fischer

Tag und Ort der Einreichung: 30.11.2002, Dresden

Ich versichere, dass ich diese Diplomarbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und Zitate als solche gekennzeichnet habe.

Der zuständige Betreuer hat die Vollständigkeit der Arbeit geprüft und der verantwortliche Hochschullehrer wurde über die Abgabe der Arbeit informiert.

An dieser Stelle möchte ich mich bei Prof. Dr.-Ing. habil. W. J. Fischer für die Themenstellung und Unterstützung, sowie bei Herrn Dipl.-Ing. Peter König für die Betreuung während der Arbeit danken.

Ralf Hildebrandt

Betreuer: Dipl.-Ing. Peter König

Diese Diplomarbeit befasst sich mit Untersuchungen zu Konzepten zur Verbesserung der Zuverlässigkeit von Sensorsystemen. Es werden potentielle Fehlerquellen betrachtet und Ansätze untersucht, mit denen Fehler erkannt und im Idealfall toleriert werden.

Der Schwerpunkt bei diesen Untersuchungen liegt bei Konzepten zur Verbesserung der Eigensicherheit der CPU des Sensorsystems und zur Detektion von Fehlern mit Hilfe von Software-Algorithmen auf dieser CPU.

Da durch die Aufgabenstellung kein spezielles System vorgegeben ist, beziehen sich die Betrachtungen in dieser Arbeit allgemein auf Sensorsysteme. Wenn nötig wird aber mit Beispielen diese Vielfalt eingeschränkt.

In dieser Arbeit wird mehrfach auf die Ergebnisse aus dem der Diplomarbeit vorangegangenen Pflichtpraktikum eingegangen. [8]

In this work some concepts for improvements of the reliability of sensor systems are presented. Potential error sources and ideas to detect and to tolerate errors are examined.

The main part are concepts to imporve the so-called „self-safety“ of the CPU of the sensor system and ideas for software-based detection of errors using this CPU.

Because no sensor system is given by the job, all examinations in this work cover sensor systems in general. If nessecary examples are pointed out, to reduce the variety.

Often the results of a practical (5 months), done before this work by the outhor are used. [8]

Inhaltsverzeichnis

1. Einleitung **8**

2. Untersuchung und Bewertung bekannter Hardware- und Softwarelösungen **9**
2.1. Das Prinzip eines Sensorsystems . . . 9
2.2. Allgemeine Überlegungen . . . 10
2.3. Der Sensor . . . 11
2.3.1. Mögliche Fehlerarten des Sensors . . . 12
2.3.2. Spaciale Redundanz . . . 12
2.3.2.1. 2 Sensoren . . . 12
2.3.2.2. 3 Sensoren . . . 13
2.3.2.3. Sensor-Array . . . 14
2.3.3. Sicherheitssensor . . . 15
2.3.4. Alternative Messmethode . . . 15
2.3.5. Temporale Redundanz . . . 15
2.3.6. Stimulation des Sensors . . . 16
2.3.7. Eine Referenzquelle (Sensor-Simulator) . . . 17
2.4. Mixed-Signal - Signalverarbeitungskomponenten . . . 18
2.5. Die Datenleitungen zum Prozessor . . . 19
2.6. Fehlererkennung im Prozessor . . . 20
2.6.1. Die Kennlinienkorrektur . . . 21
2.6.2. Filtern von Spikes . . . 21
2.6.2.1. Einzelne Spikes . . . 21
2.6.2.2. Eine kurzzeitige Fehlmessung . . . 23
2.6.2.3. Ein Fehler-Burst . . . 24
2.6.2.4. Die Art der Mittelwertbildung . . . 24
2.6.2.5. Der Wertebereich bei der Mittelwertbildung . . . 25
2.6.3. Detektion von Überlast / Unterlast . . . 26
2.6.4. Bias und Drift . . . 27
2.6.5. stuck at - Fehler . . . 27
2.6.6. Erratik und Oszillation . . . 28
2.6.7. Kommunikation mit dem Host . . . 29
2.7. Zuverlässigkeit des Prozessors . . . 29
2.7.1. Alternative Berechnung . . . 30
2.7.2. Hin- und Rückrechnung . . . 30
2.7.2.1. Addition / Subtraktion . . . 31
2.7.2.2. Die logischen Funktionen . . . 31
2.7.2.3. Single Operand Instructions . . . 31
2.7.3. Fehlererkennung mit codierten Operanden . . . 31
2.7.3.1. RESO . . . 32
2.7.3.2. Rechnen mit inversen Operanden . . . 33
2.7.4. Parity . . . 36
2.7.4.1. Parity auf einem Prozessor . . . 36

2.7.4.2. Berechnung von Parity 37
2.7.4.3. MOV 38
2.7.4.4. BIS (OR) 38
2.7.4.5. AND 39
2.7.4.6. XOR 39
2.7.4.7. ADD, ADDC und JMP 40
2.7.4.8. SUB und SUBC 40
2.7.4.9. DADD 41
2.7.4.10. RRA und RRC 42
2.7.4.11. SWPB 42
2.7.4.12. SXT 42
2.7.4.13. Parity bei der Multiplikation 42
2.7.4.14. Fehlerabdeckung von Parity 42
2.7.4.15. Realisierungsmöglichkeiten von Parity-Checks 43
2.7.5. Watchdog 46
2.7.6. Fehlerschutzcodierung des Speichers 47
2.7.6.1. Der RAM des MSP430 47
2.7.6.2. Partiy 47
2.7.6.3. Hamming-Codes 48
2.7.6.4. Reed-Muller-Codes 50
2.7.6.5. BCH-Codes 50
2.7.6.6. Reed-Solomon-Codes 52
2.7.6.7. Schritte für die Implementierung im MSP430 54
2.7.7. RNS 56
2.7.8. Sicherheit der state machine 57
2.7.8.1. Die state machine als Linear Digital State Variable System 57
2.7.8.2. Eine Ablaufkontrolle für state machines 58
2.7.9. Eigentest 60
2.7.10. FPGA 61
2.8. Möglichkeiten des Bussystems 61
2.9. Erweiterungen für den Prozessor 62
2.9.1. Ein nicht-flüchtiger Speicher 63
2.9.2. Ein Fehler-Logbuch 64
2.9.2.1. Erstellung des Logbuches 65
2.9.2.2. Inhalt des Logbuches 65
2.9.2.3. Zugriffsmechanismen 67
2.9.3. Interrupt-Auslösung im Fehlerfall einer Komponente 67
2.10. Beispielsysteme 68
2.10.0.1. Siemens Moore 345 XTC 68
2.10.0.2. 1% genaue Absolutdrucksensorfamilie 68
2.10.0.3. Temperaturmessung mit einer Brückenschaltung 69
2.10.0.4. Current Window 69
2.10.0.5. IDR 69
2.10.0.6. Sensor-Stimulation eines Resonanz-Druck-Sensors 69

2.10.0.7. Eine Watchdog-Realisierung . . . 69
2.10.0.8. Ein Prozessor für den Einsatz im Weltraum . . . 70

3. Das Modellsystem Drucksensor **70**
3.1. Die Messschaltung . . . 70
3.2. Überprüfung der Brückenwiderstände . . . 71
3.2.1. Modifikation der Messbrücke . . . 71
3.2.2. Flexible Ansteuerung der herkömmlichen Messbrücke . . . 73
3.3. Ausnutzung vorhandener Redundanz . . . 74
3.3.1. Die herkömmliche Messbrücke . . . 74
3.3.1.1. Fehlerdetektion und Lokalisierung . . . 74
3.3.1.2. Wertung . . . 76
3.3.2. Eine dritte Halbbrücke . . . 76
3.3.2.1. Fehlerdetektion und Lokalisierung . . . 76
3.3.2.2. Detektierbare Fehler . . . 77
3.3.2.3. Wertung . . . 78
3.4. Rekonfiguration der herkömmlichen Messbrücke . . . 79
3.5. Das Beispielsystem . . . 80
3.5.1. Das Bussystem zum Prozessor . . . 81
3.6. Detektierbare und nicht detektierbare Fehler . . . 82

4. Umsetzung von Lösungsvorschlägen **83**
4.1. Realisierung einer RAM-Fehlerschutzkorrektur . . . 84
4.1.1. Beschreibung der Funktion der ECC-Komponente . . . 84
4.1.2. Hardwareaufwand der ECC-Komponente . . . 86
4.2. Fehlersignalisierung im MSP430 . . . 86
4.3. Systemsimulation . . . 87
4.3.1. Simulation der Messschaltung . . . 88
4.3.2. Simulation des Signalpfades . . . 89
4.3.3. Fehlerbehandlung in Software auf dem MSP430 . . . 89
4.4. Parity im MSP430 . . . 90
4.4.1. Details . . . 91
4.4.1.1. Die ALU . . . 91
4.4.1.2. Der Bus MDB_in . . . 92
4.4.1.3. Die Register . . . 92
4.4.1.4. Der Incrementer des PC . . . 92
4.4.1.5. Der Incrementer INC . . . 93
4.4.2. Fehlersignalisierung und Fehlerbehandlung . . . 93
4.4.3. Test und Testergebnisse . . . 93
4.4.3.1. Beobachtungen . . . 93
4.4.3.2. Besonderheiten . . . 94
4.4.4. Hardwareaufwand . . . 94

5. Zusammenfassung und Ausblick **95**

A. REiSO **97**
A.1. REiSO bei der Addition . . . 97
A.2. REiSO bei der Addition mit Carry . . . 98
A.3. REiSO bei der Subtraktion . . . 99
A.4. REiSO bei der Subtraktion mit Carry . . . 99
A.5. REiSO bei der Multiplikation . . . 100
A.6. RESO im Vergleich REiSO . . . 100

B. Abschätzung der maximalen Signaländerung **102**
B.1. Ein bekannter typischer Signalverlauf . . . 102
B.2. Ein Signal mit einer charakteristischen Frequenz . . . 102
B.3. Ein band- und amplitudenbegrenztes Signal . . . 103
B.4. Der Einfluss von Rauschen . . . 105

C. Lineare Block Codes **107**
C.1. Das Konzept von Block Codes . . . 107
C.2. Decodierung von Block-Codes - Syndrom-Decodierung . . . 108
C.3. Zyklische Block-Codes . . . 109
C.4. Galois Felder . . . 109

D. Die Brückenschaltung **111**
D.1. Die einfache Messbrücke . . . 111
D.2. Die dritte Halbbrücke . . . 112

E. Abkürzungen **114**

1. Einleitung

Als Sensor im elektrotechnischen Sinne wird ein System bezeichnet, welches eine nicht-elektrische Messgröße in ein elektrisches Signal wandelt. Ein Sensorsystem besteht aus einem Sensor, einer Ausleseeinheit, eventuell einer Signalverarbeitungseiheit und einer Ausgabeschnittstelle. [1]

Sensorsysteme können sowohl aus einzelnen Teilkomponenten zusammengefügt (hybrid), oder als integrierte Sensorsysteme gefertigt werden. Geringe Fertigungskosten und minimaler Platzbedarf sind Vorteile integrierter Sensorsysteme. Hybride Sensorsysteme dagegen versprechen auf die jeweilige Fertigungstechnologie optimierte Baugruppen und sie können auch für solche Messaufgaben eingesetzt werden, für die es keine Halbleitertechnologie gibt, mit der ein geeigneter Sensor in das System zu integrieren wäre.

An Sensoren und speziell an Sensorsysteme wird in erster Linie die Forderung gestellt, einen möglichst linearen Zusammenhang zwischen Messgröße und Ausgabesignal herzustellen. Eine hohe Aussteuerbarkeit, ein geringer Messfehler, eine hoher Auflösung, große Lebensdauer und auch hohe Sampling-Rate sind weitere Wünsche. Zusätzlich zu diesen grundsätzlichen Forderungen an ein Messsystem wird auch die Forderung gestellt, im Falle eines Defekts diesen mindestens zu detektieren, möglichst zu tolerieren oder aber wenigstens rudimentäre Grundfunktionen trotz der Störung bereit zu stellen.

Um die Möglichkeit eines Sensorsystems, einen Fehler zu erkennen und einen für den Benutzer sicheren Zustand anzunehmen, wir der Begriff der „Eigensicherheit“ eingeführt, wie er in [1] definiert ist:

1. Implementierte Selbsttestfunktion
   Ein Sensorsystem muss mindestens in der Lage sein, einen Fehler zu detektieren.

2. Übermittlung zusätzlicher Messinformationen
   Im Falle eines Defektes muss eine Information verfügbar sein, wie stark sich dieser Defekt auf die Messung auswirkt. Dies kann eine Information sein, dass nach dem Defekt nicht mehr gewährleistet sein kann, einen weiteren Defekt zu detektieren, wie stark sich der Defekt auf den relativen Messfehler auswirkt oder ob es zu einem Totalausfall gekommen ist. Die Minimalanforderung ist das Setzen eines simplen Fehler-Signals.

3. Messwertaufnahme mit höchster Güte unter den gegebenen Messbedingungen
   Das Sensorsystem muss alle zur Verfügung stehenden Informationen ausschöpfen, um das bestmögliche Messergebnis unter den gegebenen Umständen bereit zu stellen. Im Falle eines teilweisen Ausfalls soll ein Betrieb mit verminderter Messwertqualität (eine Art Notbetrieb) weiter möglich sein. Im Idealfall ist eine Fehlerkorrektur anzustreben.

Unterschiedliche Anforderungen an die Qualität und den Umfang dieser Eigensicherheit und die entstehenden Kosten werden eine Abstufung und Anpassung der Forderungen an ein konkretes Sensorsystem nötig machen. Eine Anwendung in einem kritischen Bereich, wie in einem Automobil macht ganz andere Sicherheitsmaßnahmen notwendig, als z. B. in einem Temperatursensor einer Wetterstation.

2. Untersuchung und Bewertung bekannter Hardware- und Softwarelösungen

Im folgenden sollen Ideen und Prinzipien zur Herstellung eigensicherer Sensorsysteme vorgestellt werden. Dabei wird besonderen Wert darauf gelegt, dass die Kosten für diese Sicherheit möglichst niedrig bleiben. Anwendungsbeispiele für extrem sicherheitskritische Anwendungen, wie bei der bemannten Raumfahrt werden daher nur am Rande betrachtet. Das Hauptaugenmerk liegt bei Sensorsystemen für den Automobilmarkt, wo einerseits die Sicherheit aller Verkehrsteilnehmer gewährleistet sein muss, aber auch die Marktwirtschaftlichkeit erhalten bleiben soll.

2.1. Das Prinzip eines Sensorsystems

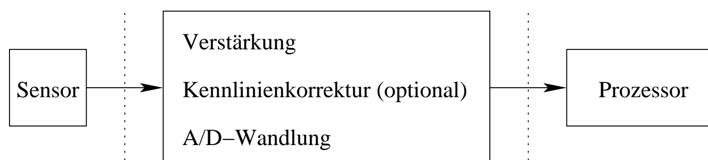


Abbildung 1: Das Prinzip eines Sensorsystems

Abbildung 1 zeigt den prinzipiellen Aufbau eines Sensorsystems. Der Sensor, der am Anfang der Signalverarbeitungskette steht, wandelt eine Messgröße in ein (analoges) elektrisches Signal. Meist wird der Anwender einen linearen Zusammenhang zwischen Messgröße und dem gewandelten Signal bevorzugen. Diese Linearität kann aber technisch nicht immer gewährleistet werden. Meist kann die Sensorkennline aber als stückweise linear angenommen werden.

Manchmal ist ein linearer Zusammenhang zwischen Messgröße und elektrischem Signal nicht herzustellen oder sogar nicht erwünscht. Eine nichtlineare Kennlinie kann z. B. von Nutzen sein, wenn in einem Teil des Messbereichs mit einer höheren Genauigkeit und ein anderen mit einer verminderten Genauigkeit gemessen werden soll.

Aus welchem Grund auch immer eine nichtlineare Sensorkennline auftritt - sie stellt eine hohe Anforderung an den A/D-Wandler. Ist die Nichtlinearität unerwünscht oder überfordert sie den eingesetzten A/D-Wandler, kann man versuchen, die Nichtlinearität vor der A/D-Wandlung zu kompensieren. In [1] werden mehrere Ansätze dafür vorgestellt.

Das analoge Sensor-Signal wird im A/D-Wandler in ein geeignetes digitales Signal gewandelt. Es gibt viele Verfahren, nach denen ein solcher A/D-Wandler arbeiten kann [2]. Der digitale Ausgang des A/D-Wandlers kann sehr unterschiedlich aufgebaut sein. Auf der einen Seite existiert die Möglichkeit ein ganzes Sampling-Wort Bit-parallel auszugeben, auf der anderen Seite ist auch eine rein serielle Ausgabe denkbar. Mischkonzepte und Prinzipien zur Erhöhung der Auflösung des A/D-Wandlers, wie in [3] mit Hilfe einer integrierten RSD-Korrektur sind ebenfalls möglich.

Konnte vor der A/D-Wandlung keine Kennlinienkorrektur durchgeführt werden, so besteht direkt nach der A/D-Wandlung noch die Möglichkeit dazu. Prinzipiell könnte man jedem möglichen digitalisierten Messwert einen korrigierten Wert über eine Tabelle zuweisen. Bei höheren Auflösungen ist diese Vorgehensweise allerdings sehr speicherintensiv. Eine andere Möglichkeit bietet der Einsatz eines Microcontrollers, der mit Hilfe einer Kennlinienapproximation (z. B. stückweise linear oder mittels geeignetem Polynom) den korrigierten Wert errechnet.

In vielen Fällen wird sich in der Signalverarbeitungskette ein DSP oder Microcontroller anschließen, der vielfältige Aufgaben zur Signalaufbereitung übernehmen kann. Filteroperationen, Extrapolationen, Sortierung, Kompression und Weitergabe der Daten über ein Bussystem sind denkbare Anwendungsgebiete. Eine (digitale) Kennlinienkorrektur kann selbstverständlich dann auch in diesem Prozessor realisiert werden.

Solch ein Prozessor, der das System zu einem „intelligenten Sensorsystem“ macht, bietet zudem weitgehende Möglichkeiten, Fehler im Sensorsystem zu detektieren und geeignet darauf zu reagieren. Die Nutzung dieser Möglichkeiten zur Erhöhung der Eigensicherheit macht einen großen Teil der Untersuchungen in dieser Arbeit aus.

Ist das Sensorsystem nicht integriert, sondern hybrid aufgebaut, so kann man oft das System in 3 Teilkomponenten aufteilen, wie in Abbildung 1 angedeutet. Diese Teilkomponenten sind der eigentliche Sensor, die Baugruppe zur A/D-Wandlung und der abschließende Microcontroller. Bezüglich der Zuverlässigkeit eines solchen hybriden Systems sind die Verbindungen zwischen den Baugruppen von Bedeutung. Ein Kontaktabriss sollte möglichst unwahrscheinlich sein, aber, wenn er doch auftreten sollte, auch sicher detektiert werden.

2.2. Allgemeine Überlegungen

Für ein zuverlässiges System gilt natürlich in besonderer Weise die Forderung nach einer qualitativ hochwertigen Fertigung und der Wunsch, bereits erprobte und ausgereifte Technologien einzusetzen. Auf diese Dinge soll aber in dieser Arbeit nicht weiter eingegangen werden, da sie zum Themenbereich der Qualitätssicherung gehören.

Damit ein System verlässlich arbeiten kann, ist auf jeden Fall eine hinreichend stabile Betriebsspannung erforderlich. Besonders analoge Baugruppen werden große Fehler verursachen, wenn die Spannung zusammenbrechen sollte. Digitale Komponenten sind oft etwas robuster, aber eine Schaltung, die mit einer für die Technologie hohen Taktfrequenz betrieben wird, ist ebenfalls anfällig für Fehler. Daher sollte eine Schaltung implementiert werden, die eine Aussage treffen kann, ob die Spannung den Anforderungen entspricht. Dies wird in erster Linie eine Schwellwertentscheidung sein. Es stellt sich das Problem, dass für eine solche Entscheidung eine Schaltung, die relativ unabhängig von der Betriebsspannung ist, konstruiert werden muss. Oft kann das realisiert werden, wenn ein geeignet dimensionierter Puffer-Kondensator eingesetzt wird. Sollte die Kondensator-Spannung groß gegenüber der Betriebsspannung werden, so kann ein Warnsignal gesetzt werden. Mit Hilfe dieses Warnsignales sollten alle laufenden Operationen gestoppt werden. Ist der Puffer-Kondensator ausreichend groß, kann ein Microcontroller noch eine

letzte Warnung über einen angeschlossenen Bus zu einem angeschlossenen Host abset-zen.

Ist der Einsatz eines solchen Puffer-Kondensators unmöglich, wäre es noch denkbar, über einen angeschlossenen Bus parasitär Energie zu entnehmen. Ob dies möglich ist, hängt von dem verwendeten Bus-System ab. Ob allerdings damit genügend Energie für den Prozessor bereit steht, um eine letzte Warnung auszugeben, ist zu bezweifeln.

Gerade bei Transponder-Systemen wird es schwer werden, noch Energie für kompli-zierte Operationen zwischenzuspeichern. Hier muss man die Funktionen auf das absolute Minimum reduzieren und notfalls den Transponder einfach abschalten. In einem solchen Fall muss bei dem Übertragungsprotokoll vom Transponder zum Host-Sender definiert werden, dass zusätzlich zu einem Datenwort eine Information gesendet wird, ob der Transponder überhaupt noch arbeitet, sodass der Sender dies erkennen kann. Denkbar wäre eine solche Implementierung, indem kein reguläres Datenwort so codiert ist, dass es dem gleicht, wie es bei abgeschaltetem Transponder-System auftritt.

Möchte man den Fehlerfall des Zusammenbruchs der Betriebsspannung für spätere Auswertungen speichern, muss nach einer Möglichkeit gesucht werden, in einem nicht-flüchtigen Speicher (wie EEPROM, Flash) eine paar Informations-Bits zu setzen. Ge-nerell sollte ein solcher Speicher vorhanden sein, auf den der Prozessor auch geeignet zugreifen kann, um dauerhaft den Zustand des Systems und wichtige Informationen ab-zulegen. Weitere Überlegungen dazu folgen in Kapitel 2.9.1.

Eine ganz andere Sache sei noch kurz erwähnt: Je nach Einsatzgebiet sollte ein Über-spannungsschutz (sowohl in positiver als auch in negativer Richtung) und gegebenenfalls auch eine Strombegrenzung integriert werden. In einem normalen Sensorsystem sollten zwar alle Baugruppen mit der selben Versorgungsspannung betrieben werden könnten und es sollte daher keine Gefährdung untereinander möglich sein, aber unter Umständen könnte eine externe Quelle z. B. den Sensor elektrisch beschädigen. In diesem Fall wäre es vorteilhaft, wenn sich dieser Effekt auf den Sensor begrenzen lassen würde, sodass unbeeinträchtigt von diesem Fehler ein Fehlersignal generiert werden könnte. Je nach Sensorsystem ist zu entscheiden, wo eine günstige Stelle für solche Schutzmechanismen existiert. Bei hybriden Sensorsystemen kann dies oft an den Kontaktleitungen zwischen den Baugruppen der Fall sein.

2.3. Der Sensor

Kernstück des Gesamtsystems ist der Sensor. Er ist in erster Linie den Umwelteinflüssen ausgeliefert und muss entsprechend robust konstruiert sein. Dennoch kann er durch Über-last, Alterung, Materialdefekt oder unsachgemäßen Betrieb unter widrigen Bedingungen ausfallen. Bestimmte Arten des Ausfalls lassen sich durch Plausibilitätstest der Messwer-te detektieren, aber z. B. ein langsamer Drift des Arbeitspunktes oder eine Verkleinerung des Aussteuerbereichs sind so nicht einfach zu detektieren.

2.3.1. Mögliche Fehlerarten des Sensors

In [1] sind folgende charakteristische Fehlerarten von Sensoren beschrieben:

- Überlast / Unterlast
- Bias (Offset zur Messgröße)
- Spike („Ausreißer“)
- stuck-at (keine / geringe Änderung des Ausgangssignals)
- Erratik (unklares Verhalten / kein Zusammenhang zur Messgröße)
- Oszillation
- Drift

Die Schwierigkeit besteht darin, alle diese Fehlerarten irgendwie zu detektieren. Z. B. Überlast / Unterlast lässt sich sehr einfach mit Hilfe von Schwellwertentscheidung feststellen. Für alle anderen Fehlerarten sind kompliziertere Detektionsmechanismen nötig.

2.3.2. Spaciale Redundanz

Die einfachste Form, bei einem Ausfall eines Sensors reagieren zu können ist, den gleichen Sensor mehrfach zu verwenden.

2.3.2.1. 2 Sensoren Mittels zweier Sensoren, ist es möglich, einen Defekt eines Sensors zu detektieren. Dabei kann man davon ausgehen, dass ein Sensor defekt ist, wenn sich beide Messwerte signifikant unterscheiden. Ist man bereit, den gesamten Datenpfad doppelt auszulegen oder puffert man die Messwerte beider Sensoren mittels einer sample & hold - Schaltung zu sequentiellen Verarbeitung, so kann diese Entscheidung innerhalb des Prozessors fallen. Anderenfalls ist eine Detektion über einen Differenzverstärker, der die beiden Sensorausgangssignale verstärkt und einem nachgeschalteten Komparator möglich. Dabei tritt allerdings das Problem auf, wie das System reagieren soll, wenn nur ein einzelner Messfehler (Spike) auftritt. Ebenfalls problematisch sind die Toleranzen bei der Fertigung der Sensoren und des Differenzverstärkers. Dieser analoge Detektor muss also so dimensioniert sein, geringe Messabweichungen zwischen den Sensoren zu ignorieren.

Kann die Entscheidung innerhalb des Microcontrollers getroffen werden, kann man geeignete Algorithmen implementieren, Spikes zu filtern und Sensor-Toleranzen zu ignorieren.

Mit 2 Sensoren kann man prinzipiell den Defekt eines Sensors feststellen. Welcher Sensor defekt ist, kann man nicht sagen.

Diese Art der Redundanz wird die erwartete Lebensdauer des Systems verringern. Ein Beispiel: Sei $P_a = 0{,}1$ die Wahrscheinlichkeit für einen Ausfall eines Sensors innerhalb

einer festen Zeitspanne. Dann gilt für den Ausfall des Gesamtsystems, wenn man die Differenzentscheidung als frei von Ausfällen annimmt:

$$P_{a(ges)} = \underbrace{P_a(1-P_a)}_{\text{Ausfall genau eines Sensors}} *2 + \underbrace{P_a^2}_{\text{Ausfall beider Sensoren}} = 0{,}19 \quad (1)$$

Oder einfacher:

$$P_{a(ges)} = 1 - \underbrace{(1-P_a)^2}_{\text{kein Ausfall}} = 0{,}19 \quad (2)$$

Die Wahrscheinlichkeit eines Ausfalls hat sich damit drastisch erhöht.

Weiterhin existiert das Problem, dass es keine Möglichkeit gibt, einen gleichzeitigen gleichartigen Defekt beider Sensoren (z. B. gleichartige Beschädigung durch Überlast) zu detektieren, da die Differenz zwischen den fehlerhaften Messwerten beider Sensoren Null sein könnte.

2.3.2.2. 3 Sensoren Möchte man nach dem Ausfall eines Sensors den Betrieb des Systems weiterhin gewährleisten, so besteht die Möglichkeit 3 Sensoren einzusetzen.

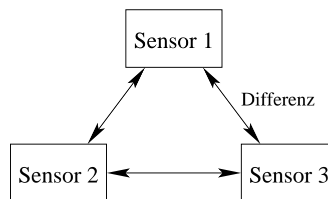


Abbildung 2: Spaciale Redundanz mit 3 Sensoren

Abbildung 2 deutet ein Schema an, in dem das Ausgangssignal jedes Sensors mit jedem anderen verglichen wird. Liefern 2 Differenzentscheidungen zu große Werte, so ist der fehlerhafte Sensor eindeutig zu lokalisieren. In diesem Fall sollte der defekte Sensor abgeschaltet werden und das System in einen 2-Sensor-Betrieb übergehen.

Ein unerwünschter Zustand könnte eintreten, wenn nur eine Differenzentscheidung einen zu großen Wert liefert. Dieser Fall wäre denkbar, wenn eine analoge Entscheidung mittels Differenzverstärker und Komparatoren realisiert wird und aufgrund von Fertigungstoleranzen die Entscheidungsschwellen nicht exakt gleich sind. Es ist daher vorteilhaft, die flexiblere digitale Lösung zu bevorzugen und dabei ausreichende Toleranzen einzukalkulieren.

An einem Beispiel soll die Ausfallwahrscheinlichkeit berechnet werden. Dabei wird angenommen, dass die Differenzentscheidung „fehlerfrei“ und hinreichend tolerant arbeitet, sodass ein Fehler genau lokalisiert werden kann. $P_a = 0{,}1$ sei wieder die Ausfallwahrscheinlichkeit für einen Sensor innerhalb einer bestimmten Zeit. Dann gilt für völlig fehlerfreien Betrieb:

$$P(0 \text{ Fehler}) = (1-P_a)^3 = 0{,}73 \quad (3)$$

Damit hat sich die Wahrscheinlichkeit für den völlig fehlerfreien Betrieb weiter verringert. Ein Defekt eines einzelnen Sensors ist dafür aber detektierbar und kompensierbar.

$$P(\text{maximal 1 Fehler}) = \underbrace{P_a(1-P_a)^2}_{\text{genau 1 Fehler}} *3 + \underbrace{(1-P_a)^3}_{\text{kein Fehler}} = 0{,}972 \quad (4)$$

Interpretiert man die Ergebnisse, so wird bei 3 Sensoren erheblich häufiger ein Fehler auftreten, als bei zwei oder nur einem Sensor. Dieser Fehler sollte aber nur mit einer Warnung gemeldet werden (sodass der Betreiber des Systems eine Reparatur in nächster Zeit eins Auge fassen sollte). Das System kann aber mit einer deutlich erhöhten Wahrscheinlichkeit weiterhin mit voller Genauigkeit betrieben werden und kann sogar noch einen Ausfall eines zweiten Sensors detektieren. Die Wahrscheinlichkeit, dass ein behebbarer Fehler und ein detektierbarer Fehler innerhalb der angenommenen Zeitspanne auftritt, berechnet sich wie folgt:

$$P(\text{genau 2 Fehler}) = P_a^2(1-P_a) = 0{,}009 \quad (5)$$

Die Gesamtausfallwahrscheinlichkeit ist die Wahrscheinlichkeit, dass das System nicht mehr benutzbar ist. Man kann sie wie folgt berechnen:

$$P_{a(ges)} = P(\text{genau 2 Fehler}) + \underbrace{P_a^3}_{\text{Ausfall aller Sensoren}} = 0{,}01 \quad (6)$$

Dabei kann man selbstverständlich wieder den gleichartigen Ausfall aller 3 Sensoren nicht mit der Differenzmethode detektieren.

2.3.2.3. Sensor-Array In einigen Fällen verwendet man nicht nur einzelne Sensoren, sondern Sensor-Arrays, wenn ein einzelner Sensor nur eine ungenügende Empfindlichkeit besitzt. [1] beschreibt ein solches System.

Ein solches Sensor-Array kann man meist partitionieren. Stellt man signifikante Unterschiede zwischen den Partitionen fest (ähnlich, wie beim Einsatz von 3 Sensoren) oder detektiert man auf anderem Wege einen Ausfall einer Partition, so sollte diese deaktiviert werden. Das dadurch abgeschwächte Sensorsignal muss anschließend mit Hilfe einer für diesen Fehlerfall angepassten Kennlinienkorrektur normalisiert werden. Damit wird selbstverständlich die Genauigkeit des Messwertes sinken, aber ein weiterer Betrieb ist problemlos möglich.

Sind Sensor-Arrays nicht räumlich stark voneinander getrennt, so ist eine gleichartige Beschädigung bei Überlast sehr wahrscheinlich. In diesem Fall kann die Differenz zwischen den Signalen der Partitionen keine Aussage über den Fehler machen.

Die Berechnungen für die Ausfallwahrscheinlichkeiten in den letzten Absätzen haben den Trend bei spacialer Redundanz gezeigt: Ein (kompensierbarer) Teilausfall wird mit steigender Anzahl von Sensoren immer wahrscheinlicher, aber der Betrieb des Systems mit voller oder zumindest ausreichender Genauigkeit ist viel besser gewährleistet.

Es ist eine Frage der Kosten und manchmal auch der Messmöglichkeiten, spaciale Redundanz in dieser Form einzusetzen.

2.3.3. Sicherheitssensor

Im Abschnitt 2.3.2 wurde immer wieder das Problem angesprochen, dass ein gleichartiger Ausfall von redundanten Sensoren nicht mit einer simplen Differenzmethode detektierbar ist. Eine Möglichkeit, dieses Manko zu beheben, ist der Einsatz eines Sicherheitssensors, wenn man Beschädigungen durch Überlast als Ausfallursache kompensieren will. Ein solcher Sensor muss einen wesentlich größeren Aussteuerbereich als der eigentliche Messsensor besitzen. Meistens haben Sensoren mit größerem Aussteuerbereich aber eine geringere Genauigkeit. Für eine einfache Schwellwertentscheidung bei Überlast spielt dies aber eine untergeordnete Rolle. Anders sieht es aus, wenn man ein 2-Sensor-System aufbauen will, wie in Kapitel 2.3.2 beschrieben und aus Kostengründen den Sicherheitssensor als zweiten Sensor verwenden möchte, um Messdifferenzen, wie sie aus Bias-Fehlern resultieren können, zu detektieren. Hier muß man in Kauf nehmen, dass dann aufgrund der erhöhten Toleranzen eine zuverlässige Aussage nur bei sehr großen Messunterschieden zwischen Sensor und Sicherheitssensor möglich ist. Auch wird dieses Vorgehen einen hohen Aufwand an eine Dimensionierung und Kalibration des Systems stellen, da Fehlentscheidungen möglichst vermieden werden sollten.

Sollte der normale Sensor als defekt detektiert werden, so sollte ein System so aufgebaut sein, dass anstelle des Signals des normalen Sensors das des Sicherheitssensors in die Signalverarbeitungskette eingespeist wird. Damit wäre ein Totalausfall des Systems abgewendet, aber die Qualität des Signals wird nur noch rudimentäre Aufgaben zulassen.

2.3.4. Alternative Messmethode

Möchte man prinzipielle Fehler einer Messmethode und Beschädigungen des Sensors, welche spezifisch für die Art der Messmethode sind kompensieren, so wäre es vorteilhaft, einen Sensor, der nach einer alternativen Messmethode arbeitet, einzusetzen.

Leider bringt diese Idee mehrere Nachteile mit sich. Als offensichtlichste Sache steht das Problem im Raum, eine alternative Messmethode zu finden. Hat man diese gefunden, so ist es nötig einen komplett neuen Sensor zu entwerfen. Will man diesen Sensor nun für Vergleichsaussagen bezüglich der anderen Sensoren, wie in Kapitel 2.3.2 beschrieben, verwenden, so müssen beide Sensorsignale normalisiert und die Kennlinien angepasst werden. Je nach Qualität der Messsignale ist dann eine Differenzentscheidung möglich. Im allgemeinen sollten Normalisierung, Anpassung und Vergleich am einfachsten digital zu realisieren sein.

2.3.5. Temporale Redundanz

Ein Problem bei spacialer Redundanz und sofortiger Entscheidung aufgrund einer Messwertdifferenz sind kurze Spikes, also „Ausreißer“ bei der Messung. Nochmaliges Messen kann dieses Problem kompensieren. Praktisch wird sich so etwas nur mit Hilfe geeigneter Software realisieren lassen. Messwerte müssen mit ihren Vorgängern verglichen werden und je nach Messsystem müssen geeignete Kriterien gefunden werden, die Spikes klar klassifizieren, sodass sie heraus gefiltert werden können. Diesem Problem widmet sich Kapitel 2.6.2.

Mit temporaler Redundanz kann man statische Defekte am Sensor nicht erkennen. Für effektive Aussagen muss zudem gewährleistet sein, dass sich die Messgröße zwischen zwei Messungen nicht zu stark ändert. Eine geeignete hohe Überabtastung ist somit nötig.

2.3.6. Stimulation des Sensors

Auf sehr elegante Weise kann man den Sensor und auch gleich noch den gesamten analogen und mixed-signal - Datenpfad testen, wenn eine Möglichkeit besteht, den Sensor zu stimulieren. Ideen und Erläuterungen zu diesem Prinzip werden in [1] detailliert vorgestellt. Hier sollen noch einmal als Überblick die wesentlichen Dinge aus dieser Arbeit zusammengefasst werden.

Eine direkte Stimulation setzt voraus, dass das Messprinzip umkehrbar ist, so z. B. kann man einen Temperatursensor durch ein Heizelement erwärmen. Anderenfalls ist es möglich, gewisse Querempfindlichkeiten von Sensoren auszunutzen. So besitzen z. B. viele Sensoren eine Temperaturabhängigkeit.

Stimuliert man einen Sensor, so überlagert sich dieses Signal mit dem eigentlich zu messenden Signal und man kann oft nur einen Offline-Test durchführen. Nutzt man eine Querempfindlichkeit aus, so kann man manchmal online testen, da die Querempfindlichkeit sowieso oft kompensiert wird. Dabei muss man selbstverständlich zum Test des Sensors das unkompensierte Signal für eine Aussage heranziehen.

Ein Offline-Test bietet die Möglichkeit, einen großen Teil des Aussteuerbereiches des Sensors zu testen. Gleichzeitig ist bei definierten Umgebungsbedingungen ein Test der Kalibration möglich.

Eine Alternative für einen Online-Test ist die Möglichkeit, ein Signal von sehr schwacher Amplitude (unterhalb des Rauschpegels) zur Stimulation zu verwenden. Um dieses Signal im Messsignal zu detektieren, verwendet man ein Matched-Filter und ein Pseudorandom-Signal zur Stimulation. Die Details sind ein Schwerpunkt von [1].

Bei Low-Power-Systemen, wie Transpondern oder Batterie-betriebenen Systemen ist eine Stimulation oft energetisch nicht möglich.

Die Stimulation des Sensors ist die einzige Möglichkeit, Fehler in einzelnen Sensoren zu detektieren und bei mehr-Sensor-Systemen gleichartige Beschädigungen auszuschließen. Sie bietet zudem die Möglichkeit auch die Qualität der Messung zu überprüfen. Des weiteren wird ein Großteil der nachfolgenden Signalverarbeitungsstufen mit getestet. Zwar kann man dann nicht sagen, welche Komponente defekt ist, aber in erster Linie interessiert der Fakt, dass ein Defekt aufgetreten ist und man geeignet darauf reagieren muss.

Viele Sensortypen sind allerdings weder direkt noch durch Querempfindlichkeiten zu stimulieren. Vor allem bei hybriden Systemen tritt dies auf. Bei integrierten Sensorsystemen ist ein detailliertes Wissen über den Aufbau des Sensors erforderlich. Beim Entwurf dieser Sensoren sollte zudem die Möglichkeit zur Stimulation schon vorgesehen werden.

Online-Stimulationstests haben den Vorteil, einen kontinuierlichen Test zu bieten und somit schnellstmöglich einen auftretenden Fehler zu detektieren. Je nach Art der

Stimulation sind Vorkehrungen notwendig, das Stimulationssignal aus dem Nutzsignal anschließend wieder heraus zu filtern. Ist dies nicht möglich, muss man einen erhöhten Messfehler akzeptieren. Online-Stimulationen nach dem Matched-Filter-Prinzip brauchen eine gewisse Zeit, bis eine Aussage getroffen werden kann. Es ist eine Periodendauer des Stimulationssignales nötig.

Offline-Tests unterbrechen den normalen Arbeitsablauf und sind in hoch verfügbaren Systemen nur selten zu realisieren. Meist ist so ein Test in solch einem System nur während einer Initialisierungsphase beim Systemstart möglich. Sollte ein Sensor aber kontinuierlich über sehr lange Zeit arbeiten, ist dies nicht sehr effizient. Man sollte daher Möglichkeiten suchen, einen periodischen Test durchzuführen bzw. genau dann einen Test auslösen, wenn die Möglichkeit dazu besteht. Beispielsweise könnte man einen Airbag-Sensor testen, wenn das Auto still steht. Für solcherart Abhängigkeiten ist detaillierteres Wissen über das Gesamtsystem und eine Implementierung auf hoher Software-Ebene zur Kommunikation zwischen verschiedenen Geräten und Sensoren nötig. Untersuchungen in dieser Richtung werden hier nicht weiter verfolgt.

2.3.7. Eine Referenzquelle (Sensor-Simulator)

Möchte man von den Möglichkeiten, die eine Stimulation von Sensoren bietet, profitieren aber ist dies technisch nicht möglich, so kann man versuchen, ein Testsignal als Referenzgröße möglichst nahe am eigentlich sensitiven Element einzuspeisen. Allgemein kann man dies als Sensor-Simulator bezeichnen, den man statt des eigentlichen Sensors an den Kopf der Signalverarbeitungskette stellt. Bei einem solchen Sensor-Simulator gelten selbstverständlich die selben Vor- und Nachteile, wie bei der direkten Sensor-Stimulation - siehe Kapitel 2.3.6.

Umso weiter man solch eine Signaleinspeisung in den Sensor integriert, desto mehr kann man vom eigentlichen Sensor mit testen. Da man aber so nie direkt die Funktion des Sensors überprüfen kann, liegt der Schwerpunkt im Test der nachgeschalteten Baugruppen. Beherzigt man das, so wird die Idee nahe liegen, zumindest bei der Initialisierung des Systems einmal den gesamten theoretisch verfügbaren Wertebereich der Signalverarbeitung „durch zu fahren“. Dies könnte z. B. durch eine geeignet dimensionierte Monoflop-Schaltung realisiert werden. Da sicher aufgrund von Fertigungstoleranzen nicht garantiert werden kann, wie schnell dieses Monoflop den Messbereich „durch fährt“, ist damit lediglich ein Plausibilitätstest, nicht aber ein kompletter Kennlinientest möglich.

Ist ein einmaliger Test nicht ausreichend, aber eine Messunterbrechung zwecks längerem Test nicht praktikabel, so kann man auch ein konstantes Referenzsignal einspeisen und nur das einmal kurz messen, sodass die Unterbrechung der normalen Messung möglichst kurz gehalten wird.

Damit ergibt sich die Möglichkeit, etwas ganz anderes zu untersuchen: Das eingespeiste konstante Signal könnte die (sinnvoll geteilte) Betriebsspannung sein. Schwankungen in der Betriebsspannung und Alterungseffekte könnten so möglicherweise detektiert werden, wenn der A/D-Wandler eine gewisse Toleranz gegenüber der Betriebsspannung besitzt. Eine Auswertung und Interpretation des gemessenen Wertes muss in Software im

Prozessor geschehen. Um eine hohe Qualität so einer Aussage zu gewährleisten, kann ein Puffer-Kondensator für die Spannungsversorgung erforderlich werden (siehe Kapitel 2.2).

Hat man den Verdacht, dass Erratik auftritt, so sollte man die Messung eines konstanten Signals eine gewisse Zeit fortsetzen. Leider kann man prinzipbedingt damit keine durch Sensordefekt ausgelöste Erratik sondern nur solche, die aus Defekten im mixed-signal Signalverarbeitungsteil resultiert, detektieren.

Besonders bei hybriden Sensorsystemen existiert das Problem, dass es zu einem Abriss der Kontakte kommen kann. Ein Sensor-Simulator ist da eine effektive Art, dies zu testen.

2.4. Mixed-Signal - Signalverarbeitungskomponenten

Zu den Mixed-Signal - Signalverarbeitungskomponenten zählen, wie in Abbildung 1 angedeutet, eine eventuelle Kennlinienkorrektur (auf analoger, mixed-signal oder digitaler Basis), eine A/D-Wandlung und gegebenenfalls eine Komponente, die den Datenstrom auf einen Bus zum Transport zum Prozessor schreibt. Letztgenannte Komponente wird in erster Linie bei hybrid aufgebauten Sensoren zu finden sein, da dort oft nur wenige Signalleitungen zur Verfügung stehen und somit oft eine serielle Übertragung gewählt wird. Bei integrierten Sensoren kann meist der Ausgang des ADC direkt an den Microcontroller angebunden werden.

Diese hier betrachteten Komponenten bieten wenig Möglichkeiten, sie direkt zu testen. Allein die sich stark unterscheidenden Möglichkeiten, die sich bei der Realisierung dieser Komponenten bieten, macht eine allgemeine Testaussage schwer. Sinnvoll erscheint die Implementierung eines Versorgungsstromtests I_{ddx}, wie z. B. in [5] vorgestellt. Für einen funktionalen Test müssten im Idealfall die Baugruppen bei ihrem Entwurf mit geeigneten Testmechanismen ausgerüstet werden, die auf Befehl (von einem Prozessor) ausgelöst werden können. Ist im Sensorsystem dagegen der Sensor direkt stimulierbar (Kapitel 2.3.6) oder ein flexibel ausgestatteter Sensor-Simulator (Kapitel 2.3.7) vorhanden, kann man meist spezielle Testmechanismen einsparen. Es ist nicht von Interesse, welche Komponente des Signalpfades defekt ist, sondern nur, ob ein Defekt vorliegt.

Es erscheint angebracht, bei dem Entwurf des A/D-Wandlers Entwurfsprinzipien einzusetzen, die Fertigungsfehler kompensieren. (Beispiel: [3], [4]) Auch wenn bei der Fertigung keine Fehler auftreten, so kann mit diesem Vorgehen ein späteres „Wegdriften“ von Komponenten des A/D-Wandlers kompensiert werden.

Einige Ideen zur Erhöhung der Zuverlässigkeit von A/D-Wandlern werden in [6] vorgestellt. Es soll hier aber nicht weiter darauf und auf A/D-Wandler-Architekturen eingegangen werden.

Legt man den Signalpfad der mixed-signal - Komponenten mehrfach aus, so bietet sich die Möglichkeit an, daß man die Signalpfade hinter den Sensoren austauschbar macht, wie Abbildung 3 andeutet. In der Abbildung werden der Einfachheit halber nur eine analoge Kennlinienkorrektur und der ADC gezeigt.

Dieses Vorgehen folgenden Vorteil: Man kann bei begründetem Verdacht, dass die Messwerte fehlerbehaftet sind, umschalten, um zu überprüfen, ob der Fehler in einem

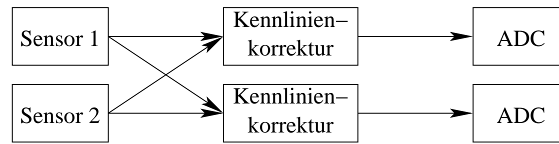


Abbildung 3: Umschalten der Signalpfade

Signalpfad liegt. Kann man so einen Fehler dort feststellen, so besteht die Möglichkeit, abwechselnd Sensor 1 und 2 über nur einen Signalpfad zu messen. Damit halbiert sich zwar die Sampling-Rate für jeden Sensor, aber ein weiterer Betrieb ist unter dieser Einschränkung möglich.

Handelt es sich bei den angeschlossenen Sensoren um redundante Ausführungen, so sollte durch das Umschalten überprüft werden, ob der Fehler, der durch eine zu hohe Differenz der Messwerte angezeigt wird, wirklich durch einen defekten Sensor ausgelöst wird und nicht etwa durch einen Fehler im Signalpfad.

Da jeder Signalpfad auch kalibriert werden muss, würden sich die Kosten für die Kalibrierung mit jedem zusätzlichen Signalpfad vervielfachen. Die Alternative dazu ist die manuelle Kalibrierung nur eines Signalpfades. Die anderen Signalpfade können dann unter definierten Testbedingungen automatisch sich selber kalibrieren, ausgehend von dem manuell kalibrierten Signalpfad als Referenz. Es ist also nur nötig eine Software zu entwickeln, die dies in Abhängigkeit von den Messwerten des manuell kalibrierten Signalpfades erledigt.

2.5. Die Datenleitungen zum Prozessor

Bei hybriden Sensorsystemen besteht die Möglichkeit eines Kontaktabrisses der Verbindungsleitungen, was zu einem stuck-at - Fehler (Festklemmen der Messwerte) führt. Einen Abriss kann man detektieren mittels Stimulation des Sensors (Kapitel 2.3.6), einem Sensor-Simulator (Kapitel 2.3.7) oder indem ein Übertragungsprotokoll verwendet wird, das diesen Fehlerfall erkennt (z. B. durch Senden einer Test-Bitfolge (z. B. $\{1, 0, 1\}$) oder wie beim I^{2}C-Bus mittels Empfangsbestätigung).

Eine Möglichkeit für eine Hardware-Detektion auf jeder beliebigen Datenleitung ist denkbar, wenn man über eine hinreichend lange Zeit die Datenleitung beobachtet und feststellt, ob eine Veränderung aufgetreten ist.

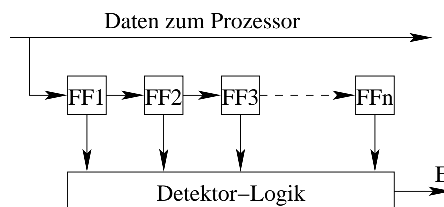


Abbildung 4: Detektor für Veränderungen auf einem Bus

Abbildung 4 zeigt eine mögliche Schaltung für einen solchen Detektor. Es handelt

sich um eine Schieberegister-Kette (hier symbolisiert durch Flip-Flops). Die Logik für den Detektor ist recht einfach:

$$E = (FF_1 \wedge FF_2 \wedge \ldots \wedge FF_n) \vee (\overline{FF_1} \wedge \overline{FF_2} \wedge \ldots \wedge \overline{FF_n}) \tag{7}$$

E wird aktiv, wenn in alle Schieberegistern ausschließlich 0 oder ausschließlich 1 steht, wie es bei einem Fehlerfall (stuck-at 0 oder stuck-at 1 auftritt). Die Länge der Register-Kette richtet sich nach den typischen Signaleigenschaften.

Hat man einen seriellen Bus und ist die Register-Kette so lang wie ein Datenwort, so kann bei binärer Codierung E nur aktiv werden, wenn entweder der minimale oder der maximale Messwert auftritt. Kann man diese Datenworte ausschließen aus dem Wertebereich der möglichen Datenworte, so hat man ein eindeutiges Fehlerkriterium gefunden.

Der Einsatz von einem Scrambler kann nicht ohne Einschränkungen zur Abhilfe genommen werden, wenn die Datenwörter bestehend aus ausschließlich Nullen oder Einsen erlaubt sind, da zufälligerweise genau die Signalfolge übertragen werden könnte, die der Scrambler in seinen Zustandsregistern gespeichert hat, sodass für die Übertragung alle Einsen ausgelöscht werden.

Bei einer parallelen Übertragungsstrecke tritt das beschriebene Problem besonders stark auf, denn falls eine binäre Codierung eingesetzt wird und alle Messwerte über einen größeren Zeitraum z. B. im unteren Teil des Aussteuerbereiches des Sensors liegen, so ist das MSB stets 0, also eine Folge von vielen Nullen auf der Datenleitung für das MSB nichts ungewöhnliches.

Eine serielle Übertragung ist generell zu empfehlen. Sie reduziert einerseits die Anzahl von Padzellen in der Schaltung, was die Möglichkeit bietet, besonders robuste (große) Verbindungen zu nutzen und sie ist leichter testbar. Ob bei der Übertragung über einen seriellen Bus noch jeweils drei zusätzliche Bits als Testsignale für den Verbindungstest angehängt werden, sollte meist keine größeren Probleme verursachen. Um einen hohen Durchsatz seriell zu erreichen, ist allerdings ein hoher Takt nötig. Unter gewissen Umständen könnte das problematisch werden, aber im allgemeinen messen Sensorsysteme nicht mit derart hoher Sampling-Rate.

Um im Fehlerfall den Betrieb des Systems weiter zu gewährleisten, muss das Bussystem doppelt ausgelegt werden.

2.6. Fehlererkennung im Prozessor

Das letzte Glied in der Signalverarbeitungskette eines intelligenten Sensorsystems stellt ein Prozessor dar. In kleinen Systemen kommt meist ein Microcontroller zum Einsatz, wenn ein hoher Aufwand für Berechnungen (Filter, Transformationen ...) betrieben werden muss, wird (eventuell zusätzlich zu einem Microcontroller) ein DSP eingesetzt.

Ein solcher Prozessor hat in erster Linie die Aufgabe, den ankommenden Datenstrom zu ordnen und über ein angeschlossenes Bussystem an einen Host weiter zu geben. Wie eben angesprochen, kommen aber mehr oder weniger aufwändige Rechenaufgaben hinzu.

Ein flexibel programmierbarer Microcontroller bietet zudem die Möglichkeit, Aufgaben auszuführen, die mit der Eigensicherheit des Sensorsystems zu tun haben. In erster

Linie sind Plausibilitätsüberprüfungen der Messwerte in Software mit Kenntnis über das Gesamtsystem und den typischen Arbeitsbereich des Sensors möglich zu realisieren. Geht man einen Schritt weiter, so kann man bei vielen Microcontrollern die Möglichkeit ausnutzen, externe Komponenten anzubinden, indem sie in den Speicherbereich eingeblendet werden („RAM-mapped“). Mit Hilfe solcher Komponenten kann man Fehlersignale verschiedenster Baugruppen sammeln und z. B. über Interrupt-Routinen eine geeignete Reaktion der Software auslösen. (Siehe dazu auch Kapitel 2.9.3) Für die Eigensicherheit des Prozessors selbst kann unter anderem ein Watchdog eingesetzt werden.

2.6.1. Die Kennlinienkorrektur

Wie in Kapitel 2.1 erwähnt, kann eine Kennlinienkorrektur in dem Prozessor mit realisiert werden. Damit bietet sich die Möglichkeit, aufgrund von Plausibilitätsgründen je nach Sensorsystem, Aussagen abzuleiten, welche „Qualität“ ein Messwert hat. Beispielsweise gibt es Sensor-Systemen mit mehreren verschiedenartigen Sensoren für unterschiedliche Messgrößen öfters Kombinationen von Messwerten, die generell nie auftreten dürfen.

Auch kann unter Umständen das Messprinzip einen Linearitätsfehler aufweisen, der in einer vergrößerten Messungenauigkeit in bestimmten Messbereichen resultiert. Dies könnte parallel zu der Messdatenübertragung signalisiert werden.

2.6.2. Filtern von Spikes

Um Spikes („Ausreißer“) der Messwerte zu unterdrücken, kann man verschiedene Möglichkeiten nutzen. Oft muss dabei vorausgesetzt werden, dass die Abtastrate weit höher als die obere Grenzfrequenz ist, mit der eine Änderung des Messwertes auftreten kann, da sonst ein starker Unterschied zwischen 2 Messwerten erlaubt ist. Weitere Überlegungen zum maximal möglichen Unterschied zwischen 2 benachbarten Messwerten befinden sich in Anhang B.

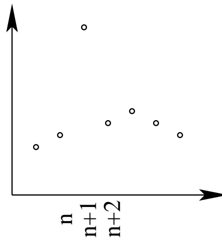


Abbildung 5: Ein einzelner Spike

2.6.2.1. Einzelne Spikes Tritt einmalig ein Spike auf, so kann dies meist recht einfach detektiert werden. Ist das Messsignal geeignet überabgetastet, so kann eine Grenzdifferenz zwischen zwei Messwerten vorgegeben werden. Wird diese Differenz (in positiver, wie negativer Richtung) überschritten, kann davon ausgegangen werden, dass ein Spike

aufgetreten sein muss. Je nach Sensorsystem gibt es verschiedene Möglichkeiten, damit umzugehen:

1. Der Spike $n + 1$ wird verworfen und der Messwert n wird stattdessen verwendet: $f(n + 1) = f(n)$.
2. Es wird der Mittelwert zwischen Wert n und Spike $n + 1$ berechnet.
3. Es wird der Mittelwert zwischen n, $n + 1$, $n + 2$, sowie gegebenenfalls zwischen weiteren benachbarten Messwerten gebildet.
4. Es wird der Mittelwert zwischen n und $n + 2$ gebildet. Der Spike findet keine Beachtung.

Mit diesen Festlegungen sind Vor- und Nachteile verbunden.

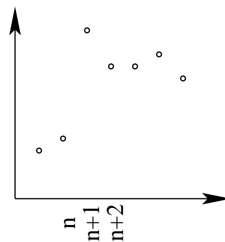


Abbildung 6: Ein einzelner Spike während eines normalen Anstieges

- Unterliegt die Messgröße einer starken, aber erlaubten Änderung (z. B. Anstieg), so kann es sein, dass der Wert $n + 2$ signifikant größer ist als der Wert n. Der Wert $n + 1$ sollte dann normalerweise zwischen beiden Messwerten liegen. Hier im Beispiel soll er aber einer Fehlmessung unterliegen und somit korrekt als Spike detektiert werden - siehe Abbildung 6. Wird nun der Spike nach Möglichkeit 1 verworfen und durch seinen Vorgänger-Wert ersetzt, so existiert wieder eine starke Differenz zwischen Wert $n + 1$ (der ja nun gleich dem Wert n ist) und dem Wert $n + 2$. Somit werden systematisch alle folgenden Werte als Spikes detektiert und das Messsystem ist unbrauchbar.
- Das eben genannte Problem lässt sich lösen, indem man Möglichkeit 2 verwendet. Dabei kann es dennoch passieren, dass bei einem Kurvenverlauf wie in Abbildung 6 der Wert $n + 2$ wieder als Spike detektiert wird, da die Differenz zwischen dem Mittelwert der Werte n und $n + 1$ und dem Wert $n + 2$ zu groß sein kann. Ist es akzeptabel, dass dann wieder der Mittelwert berechnet und an Stelle des Wertes $n + 2$ verwendet wird, so kann man ohne Modifikationen diese Vorgehensweise weiter verwenden, da in kurzer Zeit die korrigierte Messkurve sich der Originalkurve annähert und sobald die Differenz zwischen zwei Messwerten hinreichend klein ist, kein Unterschied mehr besteht.

Ist diese Verfälschung inakzeptabel, muss die Grenzdifferenz zwischen zwei Messwerten für den Messwert $n + 2$ erhöht (und danch wieder zurück gesetzt) werden.

Hat man einen Kurvenverlauf, wie in Abbildung 5, kann es störend sein, dass durch die Mittelwertbildung der Spike überhaupt in das Messergebnis einfließt. Besonders bei Regelkreisen wird unnötigerweise eine Anpassung einer Stellgröße erfolgen, die das System aus dem eigentlich bevorzugten Zustand heraus bewegt.

Ebenso kann es sein, wenn der Spike sehr deutlich vom eigentlichen Messwert abweicht, dass der berechnete Mittelwert immer noch ungewöhnlich groß ist.

- Bei geeigneter Überabtastung kann man Möglichkeit 3 verwenden. Damit fließt der Spike zwar in das Messergebnis ein, wird aber stärker unterdrückt. Diese Vorgehensweise entspricht einer starken Tiefpassfilterung.

  Problematisch bei hochpräzisen Messsystemen wird diese Vorgehensweise, da der Messfehler unter Umständen stark ansteigen kann. Mittelt man beispielsweise über 8 Messwerte und es es liegen alle Werte an der unteren Aussteuerbereichsgrenze des Sensors, so kann ein einzelner Spike, der einen Ausschlag bis zur oberen Aussteuergrenze verursacht, das Ergebnis um einen reduzierten relativen Messfehler $F_{red} = 12{,}5\%$ verfälschen.

  $$F_{red} = \frac{e_x}{x_{max} - x_{min}} = \frac{x_{falsch} - x_{richtig}}{x_{max} - x_{min}} \quad (8)$$

  Abhilfe schafft in gewissen Maßen die Mittelung über viel mehr Messwerte, welche aber eine viel höhere Überabtastung voraussetzt. Effektiv erscheint es daher, zusätzliche Plausibilitätsanalysen anzustellen. So kann der einzelne starke Ausreißer innerhalb von 7 anderen, recht gleichartigen Werten einfach erkannt werden. In diesem Falle kann er analog wie nach Möglichkeit 1 verworfen werden.

  Schon ein Vergleich von jeweils 3 benachbarten Messwerten kann einzelne Spikes recht zuverlässig detektieren. Hier würde man einen Mehrheitsentscheid machen. Problematisch ist dies, wenn nicht 2 Messwerte relativ ähnlich sind.

- Möglichkeit 4 verspricht eine nahezu ideale Elimination des Spikes unter der Voraussetzung, dass der fehlerbehaftete Messwert in Wirklichkeit tatsächlich zwischen seinen benachbarten Messwerten gelegen hat. Bei geeigneter Überabtastung sollte dies sehr häufig der Fall sein. Nachteilig ist die zusätzliche Verzögerung.

2.6.2.2. Eine kurzzeitige Fehlmessung Unter Umständen kann es vorkommen, dass nicht nur ein einzelner Messwert fehlerhaft ist, sondern dass sich der Fehler auf mehrere Abtastwerte auswirkt. Abbildung 7 zeigt einen solchen Fall.

Kann man aufgrund von detaillierten Kenntnissen über das Gesamtsystem einen solchen Fehler durch Verletzung der Plausibilität erkennen, so ist eine geeignete Filterung möglich. Schwieriger wird es in einem System, in dem es nicht möglich ist, stark überabzutasten und eine solche Folge von Messwerten auch von einer kurzzeitigen Änderung

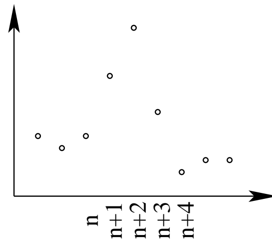


Abbildung 7: 3 fehlerhafte Messwerte

des Messsignales zum Teil mit verursacht werden kann. Kann man in einem solchen Falle aber eine derartig starke Änderung aus Plausibilitätsgründen ausschließen, so wäre ein „Abschneiden“ der Messwertspitze wünschenswert. Hier empfiehlt sich also wieder die Mittelwertbildung über mehrere Messsignale. Ob dann aber tatsächlich eine kurze Änderung im Messsignal verbunden mit einem Fehler zu diesem Fehlerbild geführt hat, oder ob es ein reiner Fehler war, kann nicht entschieden werden. Der reduzierte realtive Messfehler kann also wieder recht groß werden.

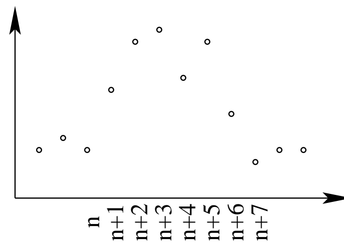


Abbildung 8: Mehrere fehlerhafte Messwerte

2.6.2.3. Ein Fehler-Burst Denkbar ist ein Fehlerfall, wie in Abbildung 8 illustriert. Da aber viele fehlerhafte Messwerte scheinbar zusammenhängen und ein solches Messergebnis durchaus auch plausibel sein könnte (wenn man nicht extreme Überabtastung einsetzt und dies so ausschließen kann), ist es denkbar schwer, geeignete Kriterien zu ermitteln, um derartige Fehler möglichst zu eliminieren. Wie im letzten Abschnitt schon angedeutet, kann eine Mittelwertbildung über mehrere Signale eine praktikable Lösung sein.

2.6.2.4. Die Art der Mittelwertbildung Bei nur 2 Messwerten wird meist der arithmetische Mittelwert gewählt, aber schon bei 3 Messwerten muss entschieden werden, welche Art der Mittelwertbildung bevorzugt wird.

- der arithmetische Mittelwert

$$x_{m_a} = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{9}$$

- der quadratische Mittelwert

$$x_{m_q} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2} \tag{10}$$

- der gleitende Mittelwert

$$x_{m_g}(N) = \frac{x_{m_g}(N-1) + x_N}{2} \qquad ; x_{m_g}(1) = x_1 \tag{11}$$

Der quadratische Mittelwert ist numerisch sehr kompliziert zu berechnen. Besonders die Wurzel verursacht einen hohen Aufwand bei der Berechnung. Gerade beim Einsatz von kleinen und verlustleistungsarmen Prozessoren hat man oft nicht die Zeit für eine derartige Rechnung.

Der arithmetische Mittelwert ist weit einfacher auf gängigen Rechnerarchitekturen zu realisieren. Die Summenbildung kann akkumulativ mit dem Eintreffen eines neuen Datenwertes geschehen. Wählt man zweckmäßigerweise $N = 2^n$ mit $n \in \mathbf{N}$, so kann die Division als shift-nach-rechts-Operation um n Stellen ausgeführt werden, was viel Rechenzeit spart. Für diese Mittelwertbildung sind also $N-1$ Summationen und eine shift-Operation um $N-1$ Stellen nötig. Damit erhält man nach N Messwerten einen Mittelwert. Möchte man dagegen nach jedem aktuellen Messwert den Mittelwert der jeweils 8 letzten Datenwörter wissen, so sind die letzten 8 Datenwörter abzuspeichern und der Mittelwert neu zu berechnen.

Will man nach jedem Messwert eine Information abrufen können, ist der gleitende Mittelwert am effizientesten zu berechnen. Es ist einfach der Mittelwert aus dem letzten Zyklus mit dem aktuellen Messwert zu addieren und das Ergebnis um eine Stelle nach rechts zu shiften. Beim gleitenden Mittelwert fließen zwar alle vorherigen Datenwerte, die je aufgetreten sind in die Rechnung ein, aber den Hauptanteil an dem Ergebnis haben die letzten 3 bis 4 Werte und der aktuelle Messwert. Somit ist beim gleitenden Mittelwert keine Flexibilität vorhanden, über eine wählbare Anzahl von Messwerten zu mitteln, wie es beim arithmetischen Mittelwert der Fall ist.

Der arithmetische Mittelwert ist zu bevorzugen, wenn eine ausreichende Überabtastung vorhanden ist und flexibel über eine größere Anzahl von Werten gemittelt werden soll. Der gleitende Mittelwert ist dagegen optimal, wenn Ergebnisse mit einer hohen Aktualität vorhanden sein sollen und wenig Rechenzeit zur Verfügung steht.

2.6.2.5. Der Wertebereich bei der Mittelwertbildung Bei der Mittelwertbildung ist darauf zu achten, dass der Wertebereich des verwendeten Zahlensystems nicht überschritten wird. Übliche Prozessoren arbeiten im Binärsystem, was bei n Binärstellen einen Wertebereich von

$$Wb_{bin} = \begin{cases} [-2^{n-1}, 2^{n-1}-1] & \text{, signed} \\ [0, 2^n - 1] & \text{, unsigned} \end{cases} \tag{12}$$

besitzt. Der Wertebereich verdoppelt sich mit jedem zusätzlichen Bit, wächst also mit 2^n.

Bei der Bildung des arithmetischen oder gleitenden Mittelwertes wird bei der Addition nur eine lineare Zunahme des Wertebereichs mit der Anzahl der aufsummierten Werte N eintreten. Stellt man sicher, dass $2^n \geq (N+1)2^k$, wobei k die Anzahl der notwendigen Binärstellen für ein Datenwort und N die Anzahl der Summationen ist, so kommt es zu keinem Zahlenbereichsüberlauf.

Bei der Berechnung des gleitenden Mittelwertes ist N selbstverständlich 1 und somit $n = k + 1$. Für den arithmetischen Mittelwert soll das folgende Beispiel das Problem verdeutlichen: Sei $n = 16$ die Wortbreite des Prozessors und $k = 12$ die Anzahl der Bits, die tatsächlich für ein Datenwort benötigt werden (incl. Vorzeichen, falls vorzeichenbehaftete Rechnung). Dann gilt

$$\frac{2^n}{2^k} \geq N + 1 \tag{13}$$

und somit muss $16 \geq N+1$ sein. Es können also 15 Additionen ohne Zahlenbereichsüberlauf ausgeführt werden. Wird $k = 13$, so sind noch 7 Additionen möglich.

2.6.3. Detektion von Überlast / Unterlast

Die Begriffe Überlast und Unterlast sollen kurz an dem Beispiel eines Drucksensors erklärt werden. Ein solcher Drucksensor wird einen bestimmten Arbeitsbereich besitzen, z. B. halber bis doppelter Normaldruck. Zu Beschädigungen am Sensor kann es kommen, wenn der Druck höher (Überlast) oder auch niedriger (Unterlast) als die Arbeitsbereichsgrenzen ist.

Generell sollte die sensitiven Elemente so dimensioniert werden, dass sie gegenüber Über- oder Unterlast in gewissem Maße robust sind. Die folgenden Überlegungen beziehen sich also im wesentlichen auf den Aussteuerbereich bei normaler Anwendung.

Der Fehlerfall Überlast bzw. Unterlast sollte im allgemeinen recht einfach zu detektieren sein. Der Messwert muss an der oberen / unteren Grenze des Aussteuerbereiches liegen. Es kann dann aber nicht unterschieden werden, ob der Messwert exakt die Dynamikgrenze erreicht hat, oder schon darüber liegt. Will man darüber ebenfalls eine Aussage treffen, sollte das gesamte System so dimensioniert werden, dass der Aussteuerbereich des A/D-Wandlers etwas größer ist, als der des normalen Arbeitsbereiches, wie Abbildung 9 illustriert. Wenn dabei nur der Aussteuerbereich des A/D-Wandlers ver-

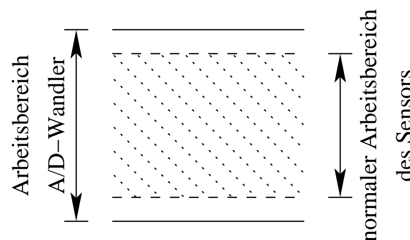


Abbildung 9: Anpassung der Aussteuerbereiches

größert wird, ohne auch dessen Auflösung zu vergrößern, ist es selbstverständlich, dass man dadurch einen größeren Messfehler erhält.

Je nach Sensorsystem und dessen Einsatzgebiet kann man Plausibilitätskriterien finden, ab wann ein Messwert außerhalb des normalen Aussteuerbreiches als kritisch anzusehen ist. Dabei kann „kritisch" bedeuten, dass die sensitiven Elemente gefährdet sind oder dass die Messwerte für das Einsatzgebiet problematisch erscheinen. Besonders einzelne Fehlmessungen, welche über den normalen Aussteuerbereich hinaus gehen, sollten im allgemeinen toleriert und kompensiert werden und nicht sofort zu einer Gesamtabschaltung wegen Überlast führen. Hier ist also ein (Software-)Zähler sinnvoll, der registriert, wieviele Messwerte (hintereinander) außerhalb des zulässigen Aussteuerbereiches liegen. Ist eine bestimmte Anzahl erreicht, kann man davon ausgehen, dass tatsächlich Überlast / Unterlast aufgetreten ist.

Eine andere Möglichkeit, Überlast / Unterlast zu detektieren, die den normalen Sensor beschädigen kann, ist der Sicherheitssensor mit vergrößertem Aussteuerbereich, wie in Kapitel 2.3.3 beschrieben. Auch hier sollte aber in intelligenter Form mit eventuellen Fehlmessungen umgegangen werden. Einzelne zu hohe / zu niedrige Messungen wurden eben schon in dem Zusammenhang erwähnt, aber auch ein genereller Defekt des Überlastsensors muss beim Systementwurf mit eingeplant werden. Besonders in Systemen mit weitreichenden Sensor-Selbsttestfunktionen, wie im besonderen Stimulation (Kapitel 2.3.6) sollte bei Überlastwarnung zuerst ein Funktionstest ausgeführt werden, um somit endgültig Überlast zu verifizieren, welche den Haupt-Sensor beschädigt haben könnte.

2.6.4. Bias und Drift

Die beiden Fehlerfälle Bias (Offset zur Messgröße) und Drift (Weggleiten der Kalibration) sind basierend auf reinen Pausibilitätsüberlegungen meist nicht detektierbar. Setzt man ein 2- oder Mehr-Sensor-System (Kapitel 2.3.2) ein, kann man durch Differenzentscheidung unter der Annahme, dass nicht alle Sensoren gleichartig betroffen sind, gegebenenfalls eine Entscheidung treffen.

Um eine bessere Aussage machen zu können empfiehlt sich wieder die Stimulation der Sensoren (Kapitel 2.3.6).

Nicht den Sensor selbst, aber die nachfolgende Signalverarbeitungskette kann man auf Drifterscheinungen mittels des Sensor-Simulators (Kapitel 2.3.7) testen.

2.6.5. stuck at - Fehler

Stuck at - Fehler (Festklemmen der Messwerte) innerhalb der Übertragungsstrecke von der mixed-signal-Signalverarbeitungseinheit zum Prozessor wurden in Kapitel 2.5 behandelt.

Will man nun stuck at - Fehler, die durch defekte sensitive Elemente im Sensor oder andere Fehlerquellen ausgelöst wurden durch Plausibilitätstest in Software detektieren, so wäre es z. B. denkbar die letzten n Datenworte zu speichern und periodisch zu überprüfen, ob eine Veränderung aufgetreten ist. Dabei gilt es folgende Probleme zu bedenken:

1. Kann es sein, dass unter regulären Bedingungen n völlig gleichartige Messwerte gemessen werden?
2. Möchte man nicht nur das absolute „Festklemmen" der Datenwerte auf genau einen Messwert detektieren, oder muss auch bei kleiner Variation des Messwertes (z. B. nur eine Änderung von der Größe des LSB) ein stuck at - Fehler angenommen werden?

Es ist zu erkennen, dass auf diese Art und Weise stuck at - Fehler in Sensorsystemen nicht eindeutig detektiert werden können. Da auch im allgemeinen keine genaueren Information über das Spektrum oder das Leistungsdichtespektrum des zu messenden Signals verfügbar sind, kann man auch keine Aussage machen, mit welcher Wahrscheinlichkeit nach m Messwerten eine Signaländerung von einer bestimmten Größe auftreten muss.

Es kann also nur durch Erfahrungswerte eine Grenze für n Messwerte, die sich jeweils maximal um x LSB unterscheiden, gezogen werden. Hat man eine solche Grenze festgelegt, so ist es nötig, mit einer anderen Prüfmethode den stuck-at - Fehler zu verifizieren. Hier bietet sich die Sensorstimulation (Kapitel 2.3.6) an.

Stuck-at - Fehler direkt am Sensor sind durch Mehr-Sensor-Anordnungen detektierbar, wenn man davon ausgeht, dass niemals alle Sensoren völlig gleichartig beschädigt werden. Dann kann ein stuck-at - Fehler nur durch Abriss aller Messleitungen zum Sensor oder durch gleichartige Beschädigung der mixed-signal Signalverarbeitungsketten (z. B. durch Ausfall der Stromversorgung) ausgelöst werden.

In Kapitel 3.3.2 wird aber eine Möglichkeit zur Detektion von Beschädigungen der Verbindungsleitungen zum Sensor gezeigt.

Ein Ausfall der mixed-signal Signalverarbeitungseinheit ist mit dem Sensor-Simulator (Kapitel 2.3.7) überprüfbar. Kann man nun wie oben beschrieben keine zufrieden stellende Aussage über stuck-at - Fehler mittels Plausibilitätstest machen, so erscheint ein zyklisches Prüfen mittels Sensor-Simulator (beispielsweise aller 100 Messwerte) eine geeignete Lösung zu sein.

2.6.6. Erratik und Oszillation

Die wohl am schwierigsten zu detektierenden Fehlerbilder sind Erratik (unklares fehlerhaftes Verhalten, kein Zusammenhang zur Messgröße) und Oszillation (Schwingungen), insbesondere, wenn diese Fehler nicht als Spike detektiert werden können (Kapitel 2.6.2). Spikes sind Fehler, wo Messwerte oberhalb der Grenzfrequenz des Messsignales auftreten. Erratik und Oszillation sind also innerhalb des erlauben Frequenzspektrums zu untersuchen.

Eine notwendige (aber bei weitem nicht hinreichende) Bedingung für die Abwesenheit von Erratik / Oszillation ist die Messung von aufeinander folgenden Messwerten, die alle gleich sind oder sich nur minimal voneinander unterscheiden. Kann also ein stuck-at-Fehler vermutet werden, ist Erratik und Oszillation sehr unwahrscheinlich - und umgekehrt. Diese Aussage ist aber sehr unbefriedigend.

Wieder liefern also nur die Sensor-Stimulation (Kapitel 2.3.6) und bei Erratik / Oszillation, welche nicht direkt im Sensor auftritt der Einsatz eines Sensor-Simulators (Kapitel 2.3.7) die einzigen hinreichenden Aussagen über diesen Fehlerfall.

2.6.7. Kommunikation mit dem Host

Wie in früheren Kapiteln schon erwähnt, kann eine effektive Erkennung von Plausibilitätsverletzungen oft nur durch zusätzliche Informationen möglich werden. Da in einem komplexen System (wie z. B. in einem Automobil) nicht alle Funktionen nur in einem einzigen Sensor-System vereint sind, existieren zwar vielfältige Abhängigkeiten untereinander, aber die Informationen stehen den intelligenten Sensoren nicht direkt zur Verfügung. In einem solchen Fall sollte im Gesamtsystem die Möglichkeit implementiert werden, die benötigten Informationen durch einen Host an die betreffenden intelligenten Sensoren zu verteilen, oder alternativ die Plausibilitätsprüfung im Host zu realisieren.

Ein Verteilen von Informationen bietet sich dagegen an, wenn der Host schon stark ausgelastet ist, die intelligenten Sensorsysteme, aber noch kleinere zusätzliche Berechnungen übernehmen können. Dabei ist zu beachten, dass das globale Bus-System dadurch nicht zu stark belastet werden sollte. Ein kontinuierliches Verteilen von Messwerten erscheint also ungeeignet. Bei einem System, in dem die einzelnen Sensoren nicht nur direkt an den Host senden können, sondern auch eine Art Broadcasting möglich ist, kann ein intelligenter Sensor immer auf dem Bus „lauschen“ und die ihn interessierenden Werte „mithören“. Sollte auch dies noch einen zu großen Traffic verursachen oder sollen einige Informationen, die sich erst durch Berechnung im Host ergeben an ein intelligentes Sensor-System gesendet werden, so muss man die Übertragung auf das nötigste reduzieren. Beispielsweise könnte in einem Automobil die Information verteilt werden, ob das Auto sich bewegt (die Räder rotieren), oder nicht. Allein diese Information reicht aus, um gewisse Schlussfolgerungen zu ziehen oder z. B. bei Stillstand des Autos (Räder stehen still, keine Bremskraft) einen (periodischen) Selbsttest zu initiieren.

2.7. Zuverlässigkeit des Prozessors

Auch wenn ein Ausfall oder Teildefekt einer analogen oder mixed-Signal-Baugruppe in typischen Sensor-Systemen wahrscheinlicher erscheint, so ist dennoch die Möglichkeit nicht auszuschließen, dass auch ein Fehler im Prozessor auftreten kann. Einerseits kann es zu Fehlern kommen, die ein falsches Rechenergebnis zur Folge haben, andererseits kann der ganze Prozessor abstürzen und einen weiteren Betrieb des Systems komplett unmöglich machen. Rechenfehler entstehen oft durch Laufzeiteffekte, wenn der Prozessor am Taktlimit betrieben wird oder durch fehlerhaftes Umschalten von Transistoren. Unter widrigen Umweltbedingungen, wie unter erhöhtem Strahlungseinfluss, wie er in der Raumfahrt auftritt, können solche Umschaltvorgänge durch ionisierende Strahlung ausgelöst werden. Untersuchungen zu fehlertoleranten Prozessoren in der Raumfahrt sind zu finden in [9], [10] und [11].

Unabhängig von den bis hierhin erwähnten transienten Fehlern existiert auch die Möglichkeit eines permanenten Fehlers. Kann man einen transienten Fehler noch durch

einfache Wiederholung der Rechnung detektieren und nach zweimaliger Wiederholung im Problemfall einen Mehrheitsentscheid machen, so sind permanente Fehler damit nicht detektierbar. Es muss nach Möglichkeiten gesucht werden, Redundanz in geeigneter Weise hinzuzufügen, sodass einerseits die Hardwarekosten niedrig bleiben, die zur Verfügung stehende Rechenzeit nicht überschritten wird und dennoch ein breites Spektrum an Fehlern erkannt und behoben werden kann. Prinzipiell könnte zwar auch beim Prozessor spaciale Redundanz (im Extremfall 3 verschiedene redundante Prozessoren) genutzt werden, aber die vergleichsweise hohen Kosten schließen diese Möglichkeit meist aus.

Der an den Prozessor angeschlossene RAM muss selbstverständlich auch betrachtet werden. Hier bieten sich Fehlerschutzcodes an, wie sie in der Nachrichtentechnik eingesetzt werden, da der RAM einem normalen Übertragungskanal gleicht.

Durch die flexible Programmierbarkeit des Prozessors ist es möglich, Ideen von spacialer Redundanz auf Lösungen über temporale Redundanz abzubilden. Da meist der Durchsatz des Prozessors nicht der limitierende Faktor in einem Sensorsystem ist und sogar meist weit über den eigentlichen Anforderungen liegt, erscheint diese Möglichkeit als die kostengünstigste.

Um die Ideen zu verdeutlichen und deren Vorteile und Grenzen herauszustellen, wird im folgenden auf den Microcontroller MSP430 [7] von Texas Instruments Bezug genommen. Teilweise wird dabei auch auf den internen Aufbau eingegangen, wie er in [8] vorgestellt wird.

2.7.1. Alternative Berechnung

Eine mehrfache Berechnung eines Ergebnisses auf dem selben Rechenweg kann zwar temporale Fehler detektieren, aber ist nicht in der Lage, permanente Fehler zu entdecken. Daher liegt es nahe, zwar die selbe Rechnung auszuführen, aber mit einem völlig verschiedenartigen Rechenweg. Das Problem dabei ist, einen solchen Weg zu finden. Gängige Rechnerarchitekturen, insbesondere RISC-Prozessoren sind darauf optimiert, möglichst effizient und platzsparend zu sein. Daher überdecken sich die unterstützten Befehle in ihren Fähigkeiten nur minimal, sodass eine Suche nach alternativen Berechnungsmethoden für einzelne Assembler-Befehle oft zu keiner völlig disjunkten Lösung führt. Es erscheint daher sinnvoll, einen high-level-Ansatz zu wählen und für eine Aufgabe verschiedene Algorithmen zu entwickeln, deren Ergebnisse man vergleichen kann

2.7.2. Hin- und Rückrechnung

Eine andere Alternative, das berechnete Ergebnis zu überprüfen besteht darin, eine Rückrechnung durchzuführen, also den Rechenschritt umzukehren. Erhält man als Ergebnis dieser Rückrechnung den erwarteten Operanden, sollte das Ergebnis richtig gewesen sein. Das ganze funktioniert selbstverständlich nur unter der Voraussetzung, dass man sich alle Operanden merkt und der Befehl umkehrbar ist.

Effektiv erscheint es auch hier, wieder einen high-level-Ansatz zu wählen, mit dem man den ganzen implementierten Algorithmus umkehrt, sofern möglich. Dennoch soll

im folgenden eine Möglichkeit zur Rückrechnung für die Befehle des MSP430 angegeben werden.

2.7.2.1. Addition / Subtraktion Die Umkehrung der Addition ist die Subtraktion. Ein `ADD Rn,Rm` kann mit einem `SUB Rm,Sum` überprüft werden. Das Ergebnis der Subtraktion muss wieder `Rn` sein. Bei einem `ADDC Rn,Rm` muss man sich den Wert des Carry-Flags merken. Für die Rückrechnung setzt man das Carry-Flag manuell auf den inversen Wert, den es vor der Rechnung hatte. Danach ergibt ein `SUBC Sum,Rm` wieder `Rn` als Ergebnis.

Die Befehle `SUB` und `SUBC` sind analog überprüfbar.

Problematisch ist der Befehl `DADD`. Da in Hardware keine Umkehrung implementiert ist, besteht die einzige Möglichkeit darin, in Software eine BCD-Subtraktion oder eine Binär ⇔ BCD - Konvertierung zu implementieren. In Anbetracht des Aufwands an Rechenzeit für beide Möglichkeiten erscheint es nur in den seltensten Fällen günstig.

2.7.2.2. Die logischen Funktionen Für die Funktionen `OR` und `AND` gibt es keine Möglichkeit eine inverse Rechnung durchzuführen. Beide Funktionen sind nicht eineindeutig.

Der Befehl `XOR` ist invers zu sich selber. Damit ist zwar eine Rückrechnung möglich, aber nur in 50% aller Fälle liegen bei der Rückrechnung andere Werte an den XOR-Gattern an, sodass keine gute Fehlerstimulation erreicht wird.

2.7.2.3. Single Operand Instructions Für ein `RRC` ist ein `ADD Res,Res` eine Shift-Operation in die entgegengesetzte Richtung, sodass nur noch das Carry-Flag, dass durch das `RRC` entstanden ist, wieder auf das LSB des zurück geshifteten Ergebnisses für den Vergleich geschrieben werden muss. Das neue Carry-Flag nach der Addition muss dann noch mit dem ursprünglichen Carry-Flag vor dem `RRC` verglichen werden.

Bei einem `RRA` entfällt die Betrachtung des neuen Carry-Flags nach der Addition.

Der Befehl `SWPB` ist zu sich selbst invers, aber eine Rückrechnung detektiert nur temporale Fehler und die korrekte Funktion von Multiplexern.

Gänzlich unmöglich ist eine Rückrechnung bei `SXT`, da der Befehl nicht eineindeutig ist.

2.7.3. Fehlererkennung mit codierten Operanden

In [12] wird die Möglichkeit vorgestellt, Berechnungen auf Prozessoren mittels nochmaliger Rechnung mit codierten Operanden zu überprüfen. Die Idee dabei ist, völlig andere Operanden für die wiederholte Berechnung zu verwenden, um mittels temporaler Redundanz spaciale Fehler erkennen zu können. Dabei muss es selbstverständlich einen Decodierungsalgorithmus geben, der das Ergebnis wieder so wandeln kann, sodass es vergleichbar mit dem Ergebnis der normalen Rechnung wird.

Zur Erläuterung: Seien F die Funktion, die eigentlich berechnet werden soll, C die Codierungsfunktion und D eine Decodierungsfunktion. Dann muss, um ein Vergleichs-

kriterium zu erhalten, folgendes gelten:

$$D(F(C(x), C(y))) = F(x, y) \tag{14}$$

Wie schon in [12] angedeutet, müssen die Codierungs- und Decodierungsfunktion leicht implementierbar sein. RESO (Kapitel 2.7.3.1, [12]) und das Rechnen mit inversen Operanden (Kapitel 2.7.3.2) stellen Möglichkeiten dar, mit codierten Operanden zu rechnen

2.7.3.1. RESO RESO - Recomputing with shifted Operands [12] ist eine Alternative zu Detektion von temporalen sowie spacial begrenzten Fehlern in Prozessoren. Der Grundgedanke dabei ist, die Operanden eines Befehls um n Bit nach links zu shiften und damit die Rechnung durchzuführen. Danach ist es meist möglich, das um m Bits nach links geshiftete Ergebnis der originalen Rechnung mit dem Ergebnis der Kontrollrechnung mit den geshifteten Operanden direkt zu vergleichen. Alternativ kann man das Ergebnis der Kontrollrechnung um m Bits nach rechts verschieben.

Beispiel: Sei folgender Codeausschnitt in C gegeben und OP1, OP2, Res, Res_shift die Namen der Operanden:

```
Res = OP1 + OP2;
OP1 = OP1 << 1;
OP2 = OP2 << 1;
Res_shift = OP1 + OP2;
```

Dann kann durch

```
if (Res_shift == (Res << 1))
```

geprüft werden, ob die Rechnung korrekt ausgeführt wurde.

Voraussetzung für die Gleichheit der Ergebnisse ist dabei selbstverständlich, dass OP1 und OP2 vor der Rechnung nur einen Wertebereich von $[-(2^{n-2}), 2^{n-2} - 1]$ haben, wenn n die Anzahl der Verfügung stehenden Bits beschreibt und vorzeichenbehaftete Rechnung angenommen wird. Sie werden somit durch das Shiften nicht verfälscht. Durch die Addition kann dann ebenfalls kein Overflow entstehen.

Das shiften nach links um 1 Bit entspricht bei Binärzahlendarstellung einer Multiplikation mit 2 (bzw. der Addition mit sich selbst). Somit lassen sich schnell folgende Zusammenhänge erkennen:

$$\begin{aligned} 2a \pm 2b &= 2(a \pm b) \\ 2a * 2b &= 4ab \end{aligned} \tag{15}$$

Die BCD-Addition kann nicht auf die selbe Art und Weise überprüft werden. Hier müssen die Operanden um 4 Bit verschoben werden, da mit jeweils 4 Bit eine Dezimalzahl codiert ist. Damit entspricht das Shiften um 4 Bit einer Multiplikation mit 10. Bei einem 16 Bit - Prozessor, wie dem MSP430 macht also diese Art der Überprüfung wenig Sinn, da der begrenzte Wertebereich der BCDs schnell zu einem Overflow beim Shiften bzw. durch die Rechnung führt.

Selbstverständlich sind sämtliche logischen Funktionen einfach durch RESO überprüfbar. Hier existiert der Vorteil, dass keinerlei Abhängigkeiten zwischen den einzelnen Bits existieren (und somit kein Overflow entstehen kann, wie bei der Multiplikation) und somit nur das MSB des Ergebnisses nicht überprüft werden kann, da dieses durch das Shiften nicht mehr existiert.

Es existiert die Idee, zu untersuchen, ob sich das shiften um 1 Bit nach rechts, also „recomputing with invers shifted operands“ (REiSO), auch für Kontrollrechnungen nutzen lässt. Dies wird in Anhang A getan. Dort wird gezeigt, dass RESO stets besser geeignet ist als REiSO.

Bei einer Realisierung von RESO mittels Software muss von der wenig realistischen Annahme ausgegangen werden, dass ausschliesslich die zu verarbeitenden Daten von einem Fehler betroffen sind, aber niemals die Programmabarbeitung selber (z. B. durch fehlerhafte Berechnung einer Sprungadresse). Sollte die Programmabarbeitung selber betroffen sein, kann mit hoher Wahrscheinlichkeit angenommen werden, dass der Prozessor unsinnige Operationen ausführt, beziehungsweise abstürzt.

Damit ist klar, dass RESO sinnvollerweise in Hardware realisiert werden muss, um eine effektive Detektion von Fehlern zu gewährleisten. Dies bedeutet aber, dass die Abarbeitung eines Befehls länger dauert, wenn man den Datenpfad nicht doppelt auslegen und lediglich eine angepasste Steuerung entwickeln will. Somit ist keine Taktkompatibilität zum Original-Prozessor mehr gewährleistet.

2.7.3.2. Rechnen mit inversen Operanden Es soll hier auf eine Möglichkeit zur Realisierung auf dem Microcontroller MSP430 näher eingegangen werden. Der Befehl `XOR #-1,Rm` invertiert `Rm` und verhält sich damit in der nach Zweierkomplement codierten Binärdarstellung wie $-Rm-1$.

Um eine effektive Realisierung zu finden, werden bei double operand instructions zunächst keine Einschränkungen gemacht, nur einen Operanden oder auch beide Operanden zu invertieren. Je nachdem, was weniger Rechenzeit beansprucht, wird eine Lösung ausgewählt. Es wird gezeigt, dass die dadurch entstehenden Lösungen einen disjunkten Weg zur Berechnung des Ergebnisses einschlagen. Speziell bei allen Operationen die den Adder des MSP430 nutzen, wird durch die so gefundenen Lösungswege automatisch immer intern mit beiden invertierten Operanden gerechnet.

Im folgenden wird wieder eine gemischt arithmetische / logische Darstellung aller Gleichungen verwendet, um den Bezug zu den realen Befehlen des MSP430 besser ersichtlich zu machen.

Die Variablen a und b repräsentieren die Operanden. In den angegebenen Pseudo-Quelltexten in Assembler stehen `A` und `B` für die Speicherorte dieser Operanden. Im einfachsten Falle sind diese Orte Register. Die Variable C steht für das Carry.

Die Additionen

$$a+b=-(-a-b)=-(-b-1-a+1)=-(\overline{b}-a+1)=\overline{(\overline{b}-a+1)}+1 \qquad (16)$$

Somit kann folgender Pseudocode geschrieben werden:

`XOR #-1,B`	$\overline{b}$
`SUB A,B`	$\overline{b}-a$
`ADD #1,B`	$\overline{b}-a+1$
`XOR #-1,B`	$\overline{(\overline{b}-a)}$
`ADD #1,B`	

Selbstverständlich ist bei dieser Realisierung keine Rücksicht auf die Flags genommen worden. Ist dies erwünscht, so muss es zusätzlich implementiert werden. Der Aufwand erscheint aber im Vergleich zum Nutzen sehr hoch, sodass es selten sinnvoll sein sollte.

Analog ist der Rechenweg bei `ADDC`:

$$a+b+C=-(-a-b)+C=\overline{(\overline{b}-a+1)}+1+C \tag{17}$$

Damit unterscheidet sich die Berechnung zu der von `ADD` nur geringfügig. Es muss das Carry gesichert und mit dem letzten Befehl zum Ergebnis dazu addiert werden. Somit ergibt sich folgender Pseudocode:

`MOV SR,X`	backup C
`XOR #-1,B`	$\overline{b}$
`SUB A,B`	$\overline{b}-a$
`ADD #1,B`	$\overline{b}-a+1$
`XOR #-1,B`	$\overline{(\overline{b}-a)}$
`MOV X,SR`	restore C
`ADDC #1,B`	

Die BCD-Addition kann mit dieser Methode nicht sinnvoll geprüft werden. Einerseits ergibt ein simples bitweises Invertieren eines BCD-Wertes möglicherweise einen Wert, der in diesem Zahlenbereich nicht definiert ist und andererseits sind im MSP430 keine negativen BCD-Werte definiert, sodass weder eine angepasste Invertierung möglich ist, noch die Erkenntnisse aus der normalen Addition verwendet werden können. Sämtliche Probleme wären zwar prinzipiell in Software lösbar, aber der Aufwand erscheint viel zu groß, als das es sinnvoll wäre.

Die Subtraktionen Es ist darauf zu achten, dass durch die generelle Struktur der Subtraktion auf dem MSP430 $dst + \mathrm{NOT}(src) + 1$ der Effekt einer Invertierung eines Operanden nicht ins Leere greift.

$$b-a=-(a-b)=-(a-b-1+1)=\overline{(a+\overline{b}+1)}+1 \tag{18}$$

`XOR #-1,B`	$\overline{b}$
`OR #1,SR`	set C in status register
`ADDC A,B`	$a+\overline{b}+1$
`XOR #-1,B`	$\overline{(a+\overline{b}+1)}$
`ADD #1,B`	

Analog kann für `SUBC` folgendes hergeleitet werden:

$$b - a - C = -(a - b + C) = -(a - b - 1 + C + 1) = \overline{(a + \overline{b} + C + 1)} + 1 \qquad (19)$$

Somit kommt ein weiterer Befehl gegenüber dem Pseudocode von `SUB` hinzu:

`XOR #-1,B`	$\overline{b}$
`OR #1,SR`	set C in status register
`ADDC A,B`	$a + \overline{b} + 1$
`ADD #1,B`	$a + \overline{b} + C + 1$
`XOR #-1,B`	$\overline{(a + \overline{b} + C + 1)}$
`ADD #1,B`	

Der Befehl `CMP` ist nicht sehr effektiv zu testen, da die Flags, die durch die Subtraktion entstehen, die einzigen Ergebnisse sind. Diese sind aber nur umständlich Schritt-für-Schritt manuell zu setzen.

Die logischen Funktionen Es sollen nur die mathematischen Zusammenhänge skizziert werden, da sich diese Funktionen sehr einfach bezüglich inverser Operanden verhalten.

`AND`	$a \wedge b = \overline{\overline{a} \vee \overline{b}}$
`BIC`	$\overline{a} \wedge b = \overline{a \vee \overline{b}}$
`BIS`	$a \vee b = \overline{\overline{a} \wedge \overline{b}}$
`XOR`	$a \oplus b = (a \wedge \overline{b}) \vee (\overline{a} \wedge b)$

Die Funktion `XOR` muss nur aufgelöst werden in ihre Grundfunktion. Der Befehl `BIT` ist ebenso, wie `CMP` nicht effektiv testbar.

Single Operand Instructions Für die Befehle `RRC` und `RRA` kann sofort folgender Pseudocode angegeben werden:

RRC	RRA
`XOR #-1,A`	`XOR #-1,A`
`RRC A`	`RRA A`
`XOR #8000h,A ;MSB invertieren`	
`XOR #1,SR`	`XOR #1,SR`
`XOR #-1,A`	`XOR #-1,A`

Die Befehle `SWPB` und `SXT` sind transparent für die Invertierung des Operanden. Nach erfolgter Berechnung muss das Ergebnis nur wieder „zurück invertiert“ werden.

2.7.4. Parity

Die Idee, eine Fehlerschutzcodierung einzusetzen, wie sie in der Nachrichtentechnik verwendet wird, liegt nahe, wenn man versucht, die Zuverlässigkeit zu erhöhen. Relativ einfach wir die Vorgehensweise bei RAM, da dieser sich wie ein Übertragungskanal verhält und somit alle Erkenntnisse der Nachrichtentechnik direkt genutzt werden können. In der Tat wird ECC-RAM (error correction code - RAM) oft durch modifizierte Hamming-Codes realisiert.

Wünschenswert wäre eine Möglichkeit, auch den verwendeten Prozessor und nicht nur den angeschlossenen RAM mittels solcher Techniken zu überprüfen. Da aber aufwändige Fehlerschutzcodes problematisch sind beispielsweise bei der Berechnung einer simplen Addition, muss man die Forderungen an Fehlererkennung und Fehlerkorrektur zurücksetzen. Eine einfache Fehlererkennung ist möglich mittels Parity-Berechnung, welche auch innerhalb des Prozessors durch den gesamten Datenpfad hindurch aufrecht erhalten werden kann.

[13] und [14] erläutern einige Grundlagen zur Parity-Berechnung und zeigen diverse Modifikationsmöglichkeiten auf. Im folgenden soll in dieser Arbeit „bit-per-word: one parity per data word“ eingesetzt werden. Die Hamming-Distanz von dieser Form von Parity ist 2, sodass jeder einzelne Bit-Fehler erkannt werden kann. Eine Korrektur ist nicht möglich.

2.7.4.1. Parity auf einem Prozessor Da der einfache Parity-Check nur eine Fehlererkennung zulässt, könnte durch eine Wiederholung der Rechnung bei permanenten Fehlern kein besseres Ergebnis erzielt werden. Hier würde nur eine alternative Berechnung des gewünschten Ergebnisses mittels disjunkten Befehlen zum Ziel führen, wenn man den Fehler exakt einem Befehl des Prozessors zuordnen könnte.

Ob es sich aber tatsächlich um einen permanenten Fehler handelt, könnte man nur ermitteln, wenn man exakt den selben Befehl wiederholen würde. Da aber die Operanden häufig durch den Befehl verändert werden, ist dies im allgemeinen nicht möglich.

Wenn also ein Parity-Fehler im Prozessor aufgetreten ist, könnte man folgende alternative Vorgehensweisen verwenden:

- Der Prozessor wird als defekt angesehen. Keine weiteren Operationen werden ausgeführt. Es wird versucht, an einen angeschlossenen Host den Defekt zu melden. (Dies könnte unmöglich sein als Software-Realisierung - z. B. bei einem permanenten Fehler im program counter (PC).)

- Der Fehler wird als temporär angesehen. Es wird ein Reset ausgeführt und der Prozessor neu gestartet. Die Anzahl dieser Neustarts ist limitiert. Sollte dieser Fall häufiger als eine noch festzulegende Anzahl eintreten, wird der Prozessor als defekt angesehen.

- Bei einem Parity-Fehler wird ein spezielles Testprogramm gestartet, welches mit möglichst hoher Fehlerabdeckung die Befehle des Prozessors prüft. Tritt bei diesem Test ein Fehler auf, wird er als permanent angesehen und der Prozessor muss als

defekt markiert werden. Eventuell sollte ein zweiter Durchlauf des Testprogramms bestätigen, dass es sich tatsächlich um einen permanenten Fehler handelt. Ist kein permanenter Defekt so detektiert worden, sollte der Prozessor neu gestartet werden.

- Der Fehler wird als temporär angesehen. Da nicht ermittelt werden kann, ob ein Befehl betroffen ist, der nur für die Programmausführung nötig ist (z. B. ein Sprungbefehl) oder ob Messdaten verfälscht worden, wird der Fehler vorerst ignoriert und lediglich ein Fehlerzähler inkrementiert. Einzelne fehlerhafte Messwerte müssen für das System als nicht schwerwiegende Fehler angesehen werden, aber es wird angenommen, dass es nötig ist, dass der Prozessor weiter arbeitet und nicht erst durch einen möglicherweise langwierigen Boot- und Test-Zyklus für eine gewisse Zeitspanne ausfällt.

  Sollte ein Befehl der Programmabarbeitung durch den Fehler betroffen worden sein, so wird der Prozessor unsinnige Operationen machen. Im Idealfall resultiert daraus ein Softwareabsturz. Ein solcher Absturz wird durch einen Watchdog (Kapitel 2.7.5), der dann nicht mehr zurückgesetzt wird, aufgefangen. Der Nachteil davon ist, dass erst nach einer gewissen Zeitspanne der Watchdog-Reset ausgeführt wird und bis dahin der Prozessor unbekannte Operationen ausführt.

  Auch bei dieser Alternative ist es sinnvoll, nur eine maximale Anzahl von Parity-Fehlern zuzulassen, bis der Prozessor als defekt markiert wird. Auch sollte ein Defekt angenommen werden, wenn mehrfach ein Parity-Fehler aufgetreten ist, ohne dass der Watchdog einen Reset ausgelöst hat. (Dann ist anzunehmen, dass die Software trotz Fehler immer wieder den Watchdog-Reset ausgeführt hat.)

2.7.4.2. Berechnung von Parity Anschaulich gesprochen wird die Anzahl der Einsen in einem Datenwort gezählt. An das Datenwort wird ein zusätzliches Bit angehängt. Dieses ist 1, wenn die Anzahl der Einsen im Datenwort ungerade ist und auf gerade Parität geprüft wird - oder es ist 0, wenn in diesem Fall auf ungerade Parität geprüft wird. Ist die Anzahl der Einsen gerade, so verhält sich das Parity-Bit umgekehrt. Beispiel:

Datenwort	odd parity	even parity
0000000000000000	1	0
0000000010000000	0	1
0000100000000100	1	0

Anders ausgedrückt: Die Anzahl aller Einsen des Datenwortes zusammen mit dem Parity-Bit muss bei „odd parity“ eine ungerade Zahl, bei „even parity“ eine gerade Zahl ergeben.

Das Zählen der Einsen lässt sich recht einfach durch XOR-Verknüpfungen realisieren. Abbildung 10 illustriert dies bei einer Wortbreite von 16 Bit für einen Parity-Checker. Ein Parity-Generator ist analog aufgebaut. Für „odd parity“ muss das Ergebnis der XOR-Verknüpfung über alle 16 Bits invertiert werden, wenn man das Parity-Bit erhalten möchte, für „even parity“ nicht.

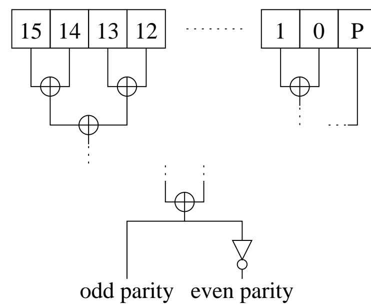


Abbildung 10: Ein Parity-Checker

Generell muss selbstverständlich einmalig festgelegt werden, mit welcher Art von Paritätsprüfung gearbeitet werden soll. Im folgenden wird daher gerade Parität („even parity") gewählt.

Es schliessen sich Untersuchungen an, wie jeder Befehl des Microcontrollers MSP430 ([7], [8]) auf das Parity-Bit reagiert. Dadurch kann eine Prüfung folgendermaßen ablaufen: Es wird ganz normal das Ergebnis eines Befehls berechnet. Vom Ergebnis wird die Parität ermittelt. Diese vergleicht man mit einer Paritäts-Vorhersage. Es gilt einen Algorithmus für die Vorhersage zu finden.

Es gelten folgende Definitionen: a und b seien Operanden im Beispiel mit einer Bitbreite von 16 Bit, $P(x)$ sei das Parity-Bit, welches zum Ausdruck x gehört bei gerader Parität. Für x kann ein einzelner Operand stehen, wie auch ein logischer Ausdruck.

2.7.4.3. MOV Da sich der Wert des Datums nicht ändert, bleibt das Parity bestehen. Eine Überprüfung ist relativ überflüssig, aber ist man gewillt, die Multiplexer der ALU zu testen, so muss man lediglich das alte Parity mit dem neuen vergleichen.

2.7.4.4. BIS (OR) Seien beide Operanden an jeder Bitstelle disjunkt zueinander, also es gilt $a \wedge b = 0$. Dann erscheint jede 1 in jedem Operanden auch im Ergebnis der OR-Verknüpfung. Es summieren sich also die Anzahl der Einsen auf. Die Anzahl der Einsen des Ergebnisses ist dann ungerade, wenn genau 1 Operand eine ungerade Anzahl von Einsen besitzt. Anders ausgedrückt:

$$P(a \vee b) = P(a) \oplus P(b) \quad | \, a \wedge b = 0 \tag{20}$$

Ist $a \wedge b \neq 0$, also gilt für eine oder mehrere Bitstellen k die Bedingung $a(k) \wedge b(k) = 1$, muss von der Summe der Einsen von a und b genau die Anzahl der Stellen, wo dies passiert, abgezogen werden. Passiert es an genau einer Stelle, so muss 1 von der Summe abgezogen werden, sodass sich das Parity-Bit des Ergebnisses umkehrt. Passiert es an 2 Stellen, so kehrt sich das Parity-Bit 2 mal um, also es hebt sich der Effekt wieder auf. Es ist also von Interesse, ob $P(a \wedge b) = 1$, denn genau dann kehrt sich das Parity-Bit des Ergebnisses um.

Damit kann man schlussendlich formulieren:

$$P(a \vee b) = P(a) \oplus P(b) \oplus P(a \wedge b) \tag{21}$$

Folgendes völlig frei gewähltes Beispiel soll das ganze noch einmal illustrieren:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	P
a	0	0	1	1	1	0	1	1	0	1	1	1	1	0	1	0	0
b	1	0	0	1	1	1	0	0	0	0	1	0	1	1	1	1	1
OR	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0
AND	0	0	0	1	1	0	0	0	0	0	1	0	1	0	1	0	1

Der Operand a besitzt 10 Einsen und damit das Parity 0, Operand b hat 9 Einsen und damit Parity 1. Die Summe der Einsen beträgt somit 19, was einem Parity von 1 entsprechen würde. An 5 Stellen besitzen beide Operanden Einsen, wie die AND-Verknüpfung zeigt, sodass im Ergebnis 14 Einsen übrig bleiben, man also Parity 0 erhält.

$$\begin{aligned} P(a \vee b) &= P(a) \oplus P(b) \oplus P(a \wedge b) \\ &= 1 \oplus 0 \oplus 1 = 0 \end{aligned}$$

2.7.4.5. AND Man kann sich die gewonnenen Erkenntnisse von der OR-Verknüpfung (Kapitel 2.7.4.4) zu nutze machen. Bekanntermaßen gilt:

$$x \oplus x = 0 \tag{22}$$

Also gilt auch:

$$y \oplus x \oplus x = y \oplus (x \oplus x) = y \oplus 0 = y \tag{23}$$

Somit folgt daraus mit (21):

$$P(a \wedge b) = P(a) \oplus P(b) \oplus P(a \vee b) \tag{24}$$

Prüft man das am Beispiel aus Kapitel 2.7.4.4 nach, so erhält man

$$\begin{aligned} P(a \wedge b) &= P(a) \oplus P(b) \oplus P(a \vee b) \\ &= 0 \oplus 0 \oplus 1 = 1 \end{aligned}$$

2.7.4.6. XOR Unter der Bedingung $a \wedge b = 0$ gelten logischerweise die selben Überlegungen, wie in Kapitel 2.7.4.4.

Gilt an einer Bitstelle k die Bedingung $a(k) \wedge b(k) = 1$, so löschen sich diese Einsen im Ergebnis der XOR-Verknüpfung aus. Die Summe der Einsen aus beiden Operanden wird also um 2 reduziert. Damit wird das Parity-Bit des Ergebnisses nicht verändert. Es gilt also:

$$P(a \oplus b) = P(a) \oplus P(b) \tag{25}$$

Ein Beispiel mit den Werten aus Kapitel 2.7.4.4 soll auch diesen Zusammenhang illustrieren.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	P
a	0	0	1	1	1	0	1	1	0	1	1	1	1	0	1	0	0
b	1	0	0	1	1	1	0	0	0	0	1	0	1	1	1	1	1
XOR	1	0	1	0	0	1	1	1	0	1	0	1	0	1	0	1	1

2.7.4.7. ADD, ADDC und JMP Das Verhalten des Parity-Bits ermittelt man am besten am repräsentativen Beispiel.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	P
a	0	0	1	1	1	0	1	1	0	1	1	1	1	0	1	0	0
b	1	0	0	1	1	1	0	0	0	0	1	0	1	1	1	1	1
carry		1	1	1					1	1	1	1	1	1			1
+	1	1	0	1	0	1	1	1	1	0	1	0	1	0	0	1	0

Es ist leicht zu erkennen, wenn man Bitposition 13 und 9 vergleicht, dass nur durch die Operanden nicht definiert ist, wie das Ergebnis-Bit aussieht. Die Carrys, die Wechselwirkungen zwischen den Bit-Spalten repräsentieren, sind für eine Aussage notwendig.

Würden die Carrys nicht existieren, würde sich das Ergebnis wie bei der XOR-Verknüpfung verhalten. An all den Stellen, wo aber ein Carry auftritt, wird das entsprechende Bit des Ergebnisses invertiert. Damit kehrt jedes Carry das Parity-Bit des Ergebnisses um. Eine gerade Anzahl von Carrys hat somit keinen Einfluss auf das Parity-Bit des Ergebnisses.

Führt man ein `ADDC` aus, so kommt ein weiteres Carry hinzu. Somit kann man schlussendlich formulieren:

$$P(a+b) = P(a) \oplus P(b) \oplus P(carry) \oplus carry_in \tag{26}$$

Hinweis: $P(carry)$ wird berechnet über alle Carrys, die sich direkt auf die 16 Bits des Ergebnisses auswirken. Das Overflow-Carry (carry-out vom MSB) gehört damit nicht dazu.

Der Befehl `JMP` ist nichts anderes als eine Addition eines Sprungoffsets zum program counter (PC) und kann damit wie eine normale Addition betrachtet werden.

2.7.4.8. SUB und SUBC Da die Subtraktion sich nur gering von der Addition unterscheidet, sind nur wenige Überlegungen nötig. Die Subtraktion im Zweierkomplement wird wie folgt ausgeführt:

$$a - b \equiv a + \bar{b} + 1 \tag{27}$$

Die Negation hat keinen Einfluss auf das Parity-Bit, da eine gerade Anzahl von Bits des Operanden vorliegt. Prozessoren mit ungerader Bitbreite sind nicht gebräuchlich.

Verwendet man statt des Befehles `SUB` den Befehl `SUBC`, so gilt folgende Gleichung:

$$a - b - \overline{C} \equiv a + \bar{b} + C \tag{28}$$

Damit gilt für die Subtraktionen der Zusammenhang analog wie in (26).

$$P(a - b - \overline{C}) = P(a) \oplus P(b) \oplus P(carry) \oplus carry_in \tag{29}$$

Beim Befehl `SUB` ist $carry_in = 1$

2.7.4.9. DADD Der Befehl `DADD` führt eine BCD-Addition aus [15]. BCD ist die Darstellung von Dezimalzahlen im normalen Binärformat mittels 4 Bit-Tupeln. Jedes 4 Bit-Tupel repräsentiert eine Dezimalstelle. Mit 4 Bit würden sich zwar $2^4 = 16$ Zahlen darstellen lassen, aber es sind nur Zahlen im Bereich $[0, 9]$ erlaubt. Bei einem 16 Bit Microcontroller, wie der MSP430 sind somit 4 Dezimalstellen darstellbar, der nutzbare Zahlenbereich ist also $[0, 9999]$.

Die BCD-Addition läuft so ab, dass ein 4 Bit-Tupel wie bei binärer Addition berechnet wird. Danach muss geprüft werden, ob das Ergebnis > 9 ist. In diesem Fall ist 6 zum Ergebnis dazu zu addieren und wenn nicht schon gesetzt, ist das carry-out des Bit 4 zu setzen. [15] veranschaulicht die Realisierung.

Mit dem Wissen von der normalen Addition kann man somit die Parity-Vorhersage ableiten. Die erste Überlegung ist, dass nach jedem 4. Bit zwar ein besonderes Carry auftritt, aber dieses Carry sich auf das Parity-Bit ebenso auswirkt, wie die anderen Carrys. Danach hat die eventuelle Addition von 6 zum Ergebnis einen Einfluss auf das Parity, da dabei ebenfalls Carrys entstehen. Da $P(6) = 0$ hat dieser Summand selber keinen Einfluss auf das Parity.

Ein repräsentatives Beispiel soll die BCD-Addition illustrieren. Es handelt sich um $1234 + 2839 = 4073$:

	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0
	0	0	1	0	1	0	0	0	0	0	1	1	1	0	0	1
mod. carry		1	1	1 †						1	1	1 †				
+	0	1	0	0	1	0	1	0	0	1	1	1	1	1	0	1
+6?					0	1	1	0					0	1	1	0
carry bei +6?					1	1							1			
$+_{BCD}$	0	1	0	0	0	0	0	0	0	1	1	1	0	0	1	1

Die mit † gekennzeichneten Carrys entstanden dadurch, dass das entsprechend niederwertige 4 Bit-Tupel nach der normalen Binäraddition > 9 ist.

Man kann also die Vorschrift für das Parity mit folgendem Pseudo-Algorithmus formulieren:

$$P(a +_{BCD} b) = P(a + b \mid \text{mod. carry nach je 4 Bit}) \oplus P(\text{carry bei} + 6) \tag{30}$$

Dabei berechnet sich der erste Teil folgendermaßen:

$$P(a + b \mid \text{modifizierte carry}) = P(a) \oplus P(b) \oplus P(carry \mid \text{modifiziert}) \oplus carry_in \tag{31}$$

Also endgültig:

$$P(a +_{BCD} b) = P(a) \oplus P(b) \oplus P(carry \mid \text{modifiziert}) \oplus P(\text{carry bei} + 6) \oplus carry_in \tag{32}$$

Hinweis: $P(\text{carry bei} + 6)$ wird über alle Carrys berechnet, die einen Einfluss auf das Endergebnis des jeweiligen 4-Tupels haben. Ein eventuelles Carry-Out bei dem MSB eines 4-Tupels gehört nicht dazu.

Abschließend soll das Ergebnis am oben angegebenen Beispiel noch einmal geprüft werden. Offensichtlich ist $P(a) = 1$ und $P(b) = 0$. Bei der binären Addition mit den

modifizierten Carrys entsteht $P(carry \,|\, \text{modifiziert}) = 0$. An 2 Stellen muss 6 zur dem jeweiligen 4 Bit-Tupel dazu addiert werden. Die dabei entstehenden Carrys haben eine Parität von $P(\text{carry bei} + 6) = 1$. Setzt man das in (32) ein, so ist $P(a +_{BCD} b) = 1 \oplus 0 \oplus 0 \oplus 1 = 0$. Dies gleicht dem Parity-Bit des Ergebnisses aus der Beispielrechnung.

2.7.4.10. RRA und RRC

Die Parity-Vorhersage ist bei `RRA` und `RRC` recht einfach: Das LSB wird zum neuen Carry-Flag. War das LSB des Operanden 1, so kehrt sich dessen Parität um. Verwendet man den Befehl `RRC`, so wird das alte Carry-Flag zum MSB des Operanden. War das Carry-Flag 1, so kehrt sich die Parität des Operanden wieder um. Beim Befehl `RRA` wird das alte MSB zum neuen MSB.

$$\begin{aligned} P(\texttt{RRA a}) &= P(a) \oplus C_{neu} \oplus MSB(a) = P(a) \oplus LSB(a) \oplus MSB(a) \\ P(\texttt{RRC a}) &= P(a) \oplus C_{neu} \oplus C_{alt} \end{aligned} \tag{33}$$

2.7.4.11. SWPB

Bei `SWPB` kann sich die Anzahl der Einsen nicht ändern, da nur die Bytes des Datenwortes getauscht werden. Somit bewirkt der Befehl keine Veränderung des Parity-Bits.

2.7.4.12. SXT

Bei dem Befehl `SXT` wird das high-byte mit dem Vorzeichen des low-byte (Bitposition 7) aufgefüllt.

Ist das Vorzeichen des low byte 0, dann verhält sich der Befehl wie (34), anderenfalls wie (35).

$$\texttt{SXT a} = (\text{00FFh}) \wedge a \qquad | \; a(7) = 0 \tag{34}$$

$$\texttt{SXT a} = (\text{FF00h}) \vee a \qquad | \; a(7) = 1 \tag{35}$$

Dementsprechend verhält sich der Befehl `SXT` bezüglich Parity ganz analog zu den Befehlen `AND` und `OR`.

2.7.4.13. Parity bei der Multiplikation

Der Multiplizierer ist eine optionale externe Baugruppe für den MSP430. Daher soll hier nur überblickshaft auf die Parity-Vorhersage in Multiplizierern eingegangen werden.

Das Parity des Ergebnisses lässt sich nur abhängig von der verwendeten Struktur des Multiplizierers vorhersagen. Prinzipiell können aber die gewonnenen Erkenntnisse aus den vorangegangenen Abschnitten genutzt werden. Basiert der Multiplizierer z. B. auf Volladdern, so kann das Carry ebenso, wie bei der normalen Addition vorhergesagt werden. [13] und [17] gehen auf die Problematik näher ein und beschreiben die Implementation der Parity-Vorhersage in verschiedenen Multiplizierer-Strukturen.

2.7.4.14. Fehlerabdeckung von Parity

Wie schon erwähnt, können mit Parity Einzelfehler detektiert werden. Damit kann man schlussfolgern, dass bei allen Befehlen, bei denen das Ergebnis von Bitposition n nicht von Bitposition m abhängt (z. B. `XOR`, `AND`), jeder Einzelfehler detektierbar ist. Jeder Einzelfehler in der Parity-Vorhersage bei diesen

Befehlen ist somit ebenfalls detketierbar, da bei der Vorhersage ebenfalls die einzelnen Vorhersage-Ergebnis-Bits unabhängig voneinander sind.

Anders sieht es bei Befehlen aus, wo oben genannte Bedingung nicht gilt, beispielsweise bei ADD. Hier ist jeder Fehler in Form eines einzelnen fehlerhaften Summen-Bits detektierbar, da sich dadurch das Parity der Summe umkehrt. Sollte aber ein carry falsch berechnet werden, so wird automatisch das Summen-Bit, das dieses fehlerhafte carry als carry_in besitzt, falsch sein. Eventuell wird durch das fehlerhafte carry_in ein fehlerhaftes carry_out ausgelöst, das sich dann analog auf die nächste Bitposition der Summe auswirkt. Da jedes fehlerhafte carry das Parity der carrys umkehrt und ein fehlerhaftes Summen-Bit auslöst, welches seinerseits wieder das Summen-Parity umkehrt, ist ein Fehler bei der Berechnung der carrys nie detektierbar.

Abhilfe kann eine separate carry-Berechnung bringen. Mit einem Vergleich der carrys, die durch den Adder erzeugt wurden mit denen aus der separaten carry-Berechnung, sind dann in einer Adder-Struktur alle carry-Fehler detektierbar.

Überträgt man dies auf den BCD-Adder, so erkennt man, dass mit dem analogen Vorgehen zwar alle normalen carrys überprüft werden können, aber kein Test der Prüfung, ob ein errechnetes 4-Tupel > 9 ist, stattfindet.

In jedem Fall bringt die separate Berechnung der carrys einen relativ hohen Hardwareaufwand mit sich. Es ist für jedes Bit ein Volladder nötig, bei dem lediglich auf ein XOR-Gatter, welches nur für die Berechnung der Summe benötigt wird, verzichtet werden kann. Da aber durch die Parity-Berechnung andererseits ebenfalls zusätzliche XOR-Gatter benötigt werden, ist der Gesamthardwareaufwand für eine Parity-Vorhersage und ein carry-Check mittels separater carry-Berechnung fast so groß, wie eine simple Verdopplung des Adders.

2.7.4.15. Realisierungsmöglichkeiten von Parity-Checks

Parity-Check in Software Strebt man eine möglichst einfache und kostengünstige Realisierung an, so liegt der Gedanke nahe, die Parity-Berechnung und -Überprüfung in Software zu erledigen. Prinzipiell ist das zwar mit jedem Befehl des MSP430 möglich, aber die Umsetzung wird sehr schwierig bei allen Operationen, wo Carrys das Ergebnis-Parity-Bit beeinflussen. Selbstverständlich kann man die Carrys auch in Software noch einmal berechnen, aber der Zeitaufwand ist enorm. Annehmbar erscheint der Aufwand also nur bei den logischen Operationen und allen single-operand instructions.

Favorisiert man dennoch eine Software-Realisierung, so muss man eine Möglichkeit finden, das Parity-Bit abzuspeichern. Bei gängigen Sensor-Systemen ist es so, dass die A/D-Wandler selten eine Auflösung von mehr als effektiv 12 bis 14 Bit gewährleisten können. Damit ist bei einem 16 Bit Prozessor die Möglichkeit vorhanden, das Parity-Bit direkt im Datenwort abzuspeichern, was selbstverständlich die Gefahr eines Overflows bei der normalen Rechnung erhöht.

Kann man das Parity-Bit nicht mit ins Datenwort hinein ziehen, gilt es zu überlegen, gleich eine leistungsstärkere Fehlerschutzkorrektur zu implementieren. Da die Parity-Vorhersage in Software nur sehr eingeschränkt möglich ist, kann man sowieso meist nur

ein Parity-Bit generieren, um Daten, die in Registern oder dem RAM abgelegt werden, so zu schützen. Beim Lesen aus Registern und RAM kann man dann die Prüfung durchführen. Diese Vorgehen ist mit einem leistungsstärkeren Fehlerkorrekturcode auch möglich. Das Problem ist eigentlich nur der erhöhte Codierungsaufwand für einen guten Fehlerkorrekturcode. Auf Fehlerkorrekturcodes wird in Kapitel 2.7.6 noch näher eingegangen.

Parity-Check in Hardware Die vorherigen Kapitel haben gezeigt, dass die Parity-Bit-Berechnung und -Vorhersage einen relativ geringen Hardwareaufwand mit sich bringen. Es gilt also zu überlegen, ob eine Integration in einen Prozessor sinnvoll wäre. Die erste Voraussetzung dafür ist das Vorhandensein einer synthesefähigen Beschreibung in einer Hardwarebeschreibungssprache. Für den MSP430 wurde eine solche Beschreibung erstellt [8]. Die interne Architektur ist aufgebaut, wie in Abbildung 11 illustriert.

Abbildung 11: Datenpfad der CPU des MSP430 [8]

Es stellt kein Problem dar, sämtliche Busse und Register um 1 Bit für das Parity-Bit zu verbreitern. Für die Parity-Vorhersage sind allerdings einige Eingriffe in die ALU nötig. Zum Beispiel vom Adder müssen sämtliche Carrys abgegriffen und zur Parity-

Vorhersage hinzugezogen werden. Das durch die Parity-Vorhersage bestimmte Parity wird dann mit dem Parity vergleichen, welches sich aus dem Ergebnis der normalen Rechnung ergibt. Vorhersage und Kontrolle erfolgen immer zusammen..

Weiterhin müssen an geeigneter Stelle Parity-Checker implementiert werden, die aus dem Datenwort das Parity neu berechnen und mit dem gespeicherten Parity-Bit vergleichen.

Die Parity-Checker sollten völlig transparent für alle anderen Komponenten sein. Sollte ein Parity-Fehler auftreten, muss die betroffene Komponente die Möglichkeit besitzen, diesen Fehlerfall anzuzeigen. Denkbar wäre dafür eine einfache externe Komponente, die in den RAM-Bereich der CPU eingebettet wird („RAM-mapped“). Näheres dazu in Kapitel 2.9.

Mögliche Punkte für die Überprüfung der Parität lassen sich recht einfach finden: Die Register-Ausgänge, der Ausgang der ALU und die Ausgänge der Incrementer sind sinnvolle Stellen.

Auf Parity in der ALU wurde in den vorangegangenen Kapiteln ausführlich eingegangen. An dieser Stelle ist zweifelsfrei eine Parity-Vorhersage nötig.

Die Register, die durch Latches aufgebaut sind, sollten durch einen Parity-Check gesichert werden. Prinzipiell würden zwar die meisten Einzelfehler auch durch die Parity-Vorhersage in der ALU detektiert werden, aber da die Register auch über den src_bus oder dst_bus auf den MAB schreiben dürfen, könnten Fehler ohne bemerkt zu werden sich auf den RAM und in den RAM eingebundene Komponenten auswirken.

Möchte man man einen Fehler im dedizierten Incrementer detektieren, so ist auch an dieser Stelle eine Parity-Vorhersage für die Addition zu implementieren. Anderenfalls muss nur ein Parity-Generator eingesetzt werden, um das neue Parity, dass sich durch Addition ergeben hat, zu bestimmen.

Mit den vorgestellten Kontrollpunkten ist der gesamte Datenpfad abgesichert. Macht man Abstriche bei der Sicherheit, so kann man Kontrollpunkte unbeachtet lassen, will man weiter erhöhte Sicherheit, so kann an vielen Stellen ein zusätzlicher Parity-Check implementiert werden. Busse bieten beispielsweise Möglichkeiten dafür.

Es erscheint denkbar, derartige Modifikationen an jedem Prozessor vorzunehmen, der als synthesefähige Hardwarebeschreibung vorliegt und eine Möglichkeit bietet, den Fehler zu signalisieren. Sollte es unmöglich sein, einen Interrupt im Fehlerfall auszulösen wäre noch folgendes denkbar:

- Man reserviert einen Bereich im RAM des Prozessors, in dem man die Informationen ablegt, ob und gegebenenfalls wo ein Parity-Fehler aufgetreten ist. Es obliegt dann der Software, diese Information periodisch zu überprüfen und geeignet darauf zu reagieren. Nicht detektierbar sind z. B. Fehler in Verbindung mit program counter - Operationen, da in diesem Falle nicht gewährleistet ist, dass die Software die beschriebene Abfrage-Schleife je wieder erreicht.

- Man löst einen Reset aus. Damit ist zumindest gewährleistet, dass nicht mit fehlerhaften Daten weiter gerechnet wird, auch wenn natürlich diese „brutale“ Methode

kaum für eine differenzierte Problembewältigung geeignet ist und zusätzlich das Sensor-System für eine gewisse Zeitspanne offline geschaltet wird.

2.7.5. Watchdog

Ein Watchdog ist eine Baugruppe für Prozessoren, die nach einer gewissen Zeit einen Reset auslöst, wenn nicht durch Software der Timer des Watchdog wieder rückgesetzt wird. Somit ist es möglich zu erreichen, dass bei Absturz des Prozessors oder bei einem Software-Bug in Form einer Endlosschleife, der Prozessor automatisch wieder rückgesetzt werden kann.

Typische Intervalle, nach denen solch ein Reset ausgeführt werden könnte, liegen meistens in Größenordnungen von 1 bis 1000 Millisekunden, was für einen Prozessor eine relativ lange Zeit, aber für das Gesamtsystem oft ausreichend kurz ist, um diese Zeit ohne Gefahr zu überbrücken. Selbstverständlich ist eine Verzögerung bedingt durch den Reboot des Prozessors zu berücksichtigen, wenn man die Zeitspanne ermittelt, die der Prozessor für das Gesamt-System offline sein darf.

Bei der Entwicklung der Software für das intelligente Sensor-System ist dann vom Programmierer an geeigneten Zeitpunkten ein Timer-Reset für den Watchdog einzuflechten. Solch ein Reset sollte möglichst selten und möglichst nicht innerhalb von Schleifen ausgelöst werden, damit der Prozessor nicht in einer Schleife verharrt, in der aber ständig der Watchdog-Timer rückgesetzt wird. Günstige Punkte für einen Reset lassen sich finden, in dem man die Software möglichst modular aufbaut und somit von einem Hauptprogramm aus Prozeduren und Funktionen aufruft. Nach einem solchen Modulaufruf sollte im allgemeinen eine günstige Position für solch einen Timer-Reset sein.

Schwierig wird die Suche nach günstigen Reset-Punkten, wenn Algorithmen verwendet werden, die nicht nach definierten Zeiten enden, sondern je nach aufkommenden Daten mehr oder weniger lange dauern. Hier muss im Einzelfall ein Kompromiss zwischen Länge des Timer-Intervalls und Häufigkeit des Timer-Resets gefunden werden.

Mittels Watchdog lässt sich weder vermeiden, dass fehlerhafte Datenwerte berechnet werden, noch dass überhaupt Offline-Zeiten des Prozessors eintreten. Man kann lediglich die Wahrscheinlichkeit für einen Totalabsturz des Prozessors stark verringern (wenn auch nicht eliminieren) und somit Offline-Zeiten auf ein verträgliches Maß begrenzen.

Der Microcontroller MSP430 besitzt einen Watchdog, der auch in der synthesefähigen Beschreibung [8] existiert. Es existieren Flags, mit denen überprüft werden kann, ob ein Reset durch den Watchdog oder eine andere Komponente ausgelöst wurde, sofern während der ganzen Zeit die Betriebsspannung nicht zusammengebrochen ist. Damit kann eine angepasste Software Rückschlüsse ziehen und geeignet reagieren. Beispielsweise wäre denkbar, Watchdog-Resets in einem nicht-flüchtigen Speicher zu protokollieren, damit erkannt werden kann, ob ein generelles Problem besteht, oder zufällige Ereignisse den Absturz ausgelöst haben.

2.7.6. Fehlerschutzcodierung des Speichers

Im folgenden werden verschiedene Ansätze zur Fehlerschutzcodierung des Speichers betrachtet. Wenn möglich werden Wege gezeigt, wie eine Realisierung gefunden werden kann.

Alle vorgestellten Möglichkeiten basieren auf linearen Blockcodes. Ein kurzer Überblick dazu wird in Anhang C gegeben, detaillierte Informationen sind in [19] zu finden.

2.7.6.1. Der RAM des MSP430 Die spezielle Struktur des RAM des MSP430, wie sie in [8] im Detail beschreiben ist, macht es nötig, eine angepasste Fehlerschutzcodierung zu wählen.

Der RAM ist als Byte-orientierter RAM (8 Bit) ausgelegt, der die Fähigkeit hat, in einem einzigen Schreibzyklus ein Word (16 Bit) oder ein Byte zu schreiben bzw. zu lesen. Dabei ist es egal, ob ein Byte-Zugriff auf eine gerade oder ungerade Adresse erfolgt. Somit ergeben sich 2 Strategien, wie die Daten mit dem Fehlerschutzcode geschützt werden können:

1. Das low- und das high-byte werden voneinander getrennt codiert.
2. Es wird immer das gesamte word codiert. Damit müssen bei einem Schreibzugriff auf ein byte die Daten im jeweils anderen byte gelesen werden, um aus den gesamten 16 Bit das Codewort zu errechnen. Ein Lesen, Codieren und Zurückschreiben dieses anderen Bytes innerhalb eines normalen Schreibzyklus ist zeitlich nicht machbar. Es darf also kein Code gewählt werden, der bei der Veränderung eines Datenteiles des Codewortes den jeweils anderen Datenteil beeinflusst. Es sollten also Daten- und Prüfbits von einander getrennt sein. Ein systematischer linearer Block-Code ist eine solcher Alternative.

   Es ergeben sich daraus folgende Regeln:

   - Bei einem Schreibzugriff auf ein byte wird das jeweils andere Byte gelesen und aus so mit dem gesamten word die Prüfbits berechnet, sodass sie in einem disjunkten RAM abgelegt werden können.
   - Bei einem word-Schreibzugriff werden die Prüfbits berechnet und in einem disjunkten RAM abgelegt.
   - Bei einem Lesezugriff egal welcher Art werden stets das gesamte word plus Prüfbits gelesen, um eine Prüfung durchzuführen. Welche Daten (low oder high byte) zur CPU weitergeleitet werden, hängt dann von der Art des Lesezugriffes ab.

2.7.6.2. Partiy In Kapitel 2.7.4 wurde schon auf die Möglichkeiten, die ein simpler Parity-Check bietet, eingegangen. Die dort vorgestellten Lösungen können völlig transparent in das Interface zwischen Prozessor und RAM implementiert werden.

Die Generator-Matrix für einen einfach-Parity-Check ist nach [19]

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & \dots & 0 & 1 \\ 0 & 1 & \dots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 1 \end{bmatrix} \tag{36}$$

Da RAM typischerweise mit Wortbreiten von 2^n gefertigt wird, muss man meist auf eine Informationsstelle verzichten, wenn man Parity einsetzt. Hat man RAM mit Wortbreite $2^n + 1$ zur Verfügung, kann Parity völlig transparent eingebettet werden.

Der Große Nachteil von Parity ist die Fähigkeit, nur Einzel- (Dreifach- / Fünffach- ...) Fehler zu erkennen, aber keinen Fehler korrigieren zu können. Dafür ist die Implementierung außerordentlich simpel.

2.7.6.3. Hamming-Codes Die Hamming-Codes repräsentieren eine Gruppe von optimalen Codes mit $d_{min} = 3$, also Einfach-Fehler-Korrektur-Möglichkeit. Folgende Eigenschaften sind charakteristisch für Hamming-Codes [19]:

Blocklänge	$n = 2^c - 1$
Informationsbits	$k = 2^c - c - 1$
Anzahl der Prüfbits	$c = n - k$

Es existieren somit unter anderem ein (7,4), ein (15,11) und ein (31,26) - Hamming-Code.

Der (7,4)-Hamming-Code würde sich eignen, ein Halbbyte des RAM zu schützen. Dies bringt nahezu eine Verdopplung des RAM-Bedarfs mit sich, aber insgesamt maximal 4 Fehler können korrigiert werden. Dabei darf aber nur 1 Fehler innerhalb eines Halbbytes auftreten, was ungünstig bei Burst-Fehlern ist. Durch diesen Code kann man leicht die Daten-Bytes des RAM getrennt voneinander betrachten.

Der (15,11)-Hamming-Code hat zu wenig Informationsstellen, um ein komplettes Word des RAM zu codieren. Daher müsste für jedes Byte getrennt dieser Code eingesetzt werden. Dies würde wieder eine Verdopplung des RAM-Bedarfs bedeuten, aber nur 2 Fehler könnten erkannt werden.

Der (31,26)-Hamming-Code könnte ein komplettes Word aufnehmen, bietet aber nur die Möglichkeit zur Korrektur von nur einem Fehler.

Die Konstruktionsvorschrift für einen Hamming-Code ist recht simpel: Sei c die Anzahl der Prüfbits. Die Prüfmatrix $\mathbf{H}$ wird konstruiert, indem man alle binären c-Tupel (außer dem Null-c-Tupel) als Spalten der Prüfmatrix in beliebiger Reihenfolge verwendet. Für den (7,4)-Hamming-Code erhält man so nach [19]

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{37}$$

und

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{38}$$

Eine andere Alternative, einen geeigneten Code zu konstruieren, bringt die Möglichkeit, einen Code zu verkürzen. Dies soll am Beispiel des (15,11)-Hamming-Code verdeutlicht werden. Da der Code nur ein Byte, also 8 Bit aufnehmen muss, sind 3 Informationsbits überflüssig. Bezüglich der Generatormatrix bedeutet dies, dass 3 Spalten aus dem Informationsteil weggelassen werden können. Dies führt sofort dazu, dass auch gleichzeitig 3 Spalten entfallen. Das soll im Detail einmal verdeutlicht werden:

Die Prüfmatrix ergibt sich nach obigen Konstruktionsvorschrift:

$$\mathbf{H}_{\text{SEF}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{39}$$

Somit erhält man **G** zu

$$\mathbf{G}_{\text{SEF}} = \begin{bmatrix} 1 & & & & & & & & & & & 0 & 0 & 1 & 1 \\ & 1 & & & & & & & & & & 0 & 1 & 0 & 1 \\ & & 1 & & & & & & & & & 0 & 1 & 1 & 0 \\ & & & 1 & & & & & & & & 0 & 1 & 1 & 1 \\ & & & & 1 & & & & & & & 1 & 0 & 0 & 1 \\ & & & & & 1 & & & & & & 1 & 0 & 1 & 0 \\ & & & & & & 1 & & & & & 1 & 0 & 1 & 1 \\ & & & & & & & 1 & & & & 1 & 1 & 0 & 0 \\ & & & & & & & & 1 & & & 1 & 1 & 0 & 1 \\ & & & & & & & & & 1 & & 1 & 1 & 1 & 0 \\ & & & & & & & & & & 1 & 1 & 1 & 1 & 1 \\ & & & & & & & & \uparrow & \uparrow & \uparrow & & & & \end{bmatrix} \tag{40}$$

Die 3 (willkürlich ausgewählten) markierten Spalten werden nicht benötigt, um ein 8 Bit Wort zu codieren. Sie entfallen. Damit entfallen auch gleichzeitig die letzten 3 Zeilen. Es entsteht somit

$$\mathbf{G}_{\text{kurz}} = \begin{bmatrix} 1 & & & & & & & & 0 & 0 & 1 & 1 \\ & 1 & & & & & & & 0 & 1 & 0 & 1 \\ & & 1 & & & & & & 0 & 1 & 1 & 0 \\ & & & 1 & & & & & 0 & 1 & 1 & 1 \\ & & & & 1 & & & & 1 & 0 & 0 & 1 \\ & & & & & 1 & & & 1 & 0 & 1 & 0 \\ & & & & & & 1 & & 1 & 0 & 1 & 1 \\ & & & & & & & 1 & 1 & 1 & 0 & 0 \end{bmatrix} \tag{41}$$

$$\mathbf{H}_{\text{kurz}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & & & \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & & 1 & & \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & & & 1 & \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & & & & 1 \end{bmatrix} \tag{42}$$

Damit ist ein Code entstanden, der mindestens die minimale Hamming-Distanz von 3 besitzt, also mindestens 1 Fehler in einem Daten-Byte korrigieren kann. Es handelt sich selbstverständlich nicht mehr um einen optimalen Code.

Es kann leicht überprüft werden, dass alle Einzelfehler ein disjunktes Syndrom hervorrufen. Betrachtet man nur Fehler innerhalb des Datenteils des Codewortes, existieren also 8 mögliche korrigierbare Fehler.

2.7.6.4. Reed-Muller-Codes Reed-Muller-Codes haben die Blocklänge $n = 2^m$ und die so genannte Ordnung r', $0 \leq r' < m$. Mögliche (n,k)-Kombinationen, generiert durch m und r' sind in [19] zu finden. Bezeichnet werden diese Codes durch R(r',m). Von Interesse ist ein R(1,3) und ein R(2,5)-Reed-Muller-Code, welche ein (8,4)- und ein (32,16)-Code sind.

Die Minimaldistanz von Reed-Muller-Codes ist $d_{\text{min}} = 2^{m-r'}$.

Der R(1,3)-Reed-Muller-Code, der ein (8,4)-Code ist, würde sich dazu eignen, ein Halbbyte zu codieren. Der Doppelte RAM-Aufwand wäre der Preis für ein $d_{\text{min}} = 2^{3-1} = 2^2 = 4$. Damit können ein Fehler in einem Halbbyte korrigiert oder 3 Fehler erkannt werden. Insgesamt sind also bis zu 4 Fehler in einem Word korrigierbar, die aber nur verteilt auf die 4 Halbbytes auftreten dürfen.

Der R(2,5)-Reed-Muller-Code, der ein (32,16)-Code ist, könnte ein ganzes Word auf einmal codieren. Mit $d_{\text{min}} = 2^{5-2} = 2^3 = 8$ wären 3 Fehler korrigierbar. Dabei ist es egal, wo diese Fehler auftreten.

2.7.6.5. BCH-Codes BCH-Codes (nach ihren Entdeckern Bose, Chaudhuri und Hocquenghem) gehören zu der Gruppe der zyklischen Codes. Das Generator-Polynom kann aus dem Produkt minimaler Polynome über GF(q^s) gebildet werden. Details sind nachzulesen in [19].

BCH-Codes werden nach der Vorgabe der Anzahl korrigierbarer Fehler t_d aufgestellt. Folgende Eigenschaften charakterisieren binary narrow-sense primitive BCH-Codes mit $s \geq 3$, $t_d < 2^{s-1}$:

Blocklänge	$n = 2^s - 1$
Anzahl Prüfbits	$c = (n - k) \leq st_d$
Minimaldistanz	$d_{\text{min}} \geq 2t_d + 1$

Ein für den RAM des MSP430 geeigneter (31,16)-BCH-Code mit der vorgegebenen minimalen Distanz von 7 (was bedeutet, dass 3 Fehler korrigierbar sind) ist in [19] angegeben:

$$g(x) = x^{15} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^5 + x^3 + x^2 + x + 1 \qquad (43)$$

Die entsprechende Generatormatrix ergibts sich also zu

$$\mathbf{G} = \begin{bmatrix}
1&1&1&1&0&1&0&1&1&1&1&1&0&0&0&1&0&0&\cdots&0&0&0\\
0&1&1&1&1&0&1&0&1&1&1&1&1&0&0&0&1&0&&0&0&0\\
0&0&1&1&1&1&0&1&0&1&1&1&1&1&0&0&0&1&&0&0&0\\
0&0&0&1&1&1&1&0&1&0&1&1&1&1&1&0&0&0&&0&0&0\\
0&0&0&0&1&1&1&1&0&1&0&1&1&1&1&1&0&0&&0&0&0\\
0&0&0&0&0&1&1&1&1&0&1&0&1&1&1&1&1&0&&0&0&0\\
0&0&0&0&0&0&1&1&1&1&0&1&0&1&1&1&1&1&&0&0&0\\
0&0&0&0&0&0&0&1&1&1&1&0&1&0&1&1&1&1&&0&0&0\\
0&0&0&0&0&0&0&0&1&1&1&1&0&1&0&1&1&1&&0&0&0\\
0&0&0&0&0&0&0&0&0&1&1&1&1&0&1&0&1&1&&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&1&0&1&&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&1&0&&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&1&&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&\cdots&0&0&1
\end{bmatrix} \tag{44}$$

Die systematische Form der Generatormatrix erhält man durch Addition der Zeilen der Matrix zu

$$\mathbf{G}_{\mathrm{SEF}} = \begin{bmatrix}
1&0&\cdots&0&0&1&1&1&1&0&1&0&1&1&1&1&1&0&0&0\\
0&1&&0&0&0&1&1&1&1&0&1&0&1&1&1&1&1&0&0\\
0&0&&0&0&0&0&1&1&1&1&0&1&0&1&1&1&1&1&0\\
0&0&&0&0&0&0&0&1&1&1&1&0&1&0&1&1&1&1&1\\
0&0&&0&0&1&1&1&1&1&0&1&0&1&0&1&0&1&1&1\\
0&0&&0&0&1&0&0&0&1&0&0&0&1&0&1&0&0&1&1\\
0&0&&0&0&1&0&1&1&0&0&0&1&1&0&1&0&0&0&1\\
0&0&&0&0&1&0&1&0&1&1&0&1&0&0&1&0&0&0&0\\
0&0&&0&0&0&1&0&1&0&1&1&0&1&0&0&1&0&0&0\\
0&0&&0&0&0&0&1&0&1&0&1&1&0&1&0&0&1&0&0\\
0&0&&0&0&0&0&0&1&0&1&0&1&1&0&1&0&0&1&0\\
0&0&&0&0&0&0&0&0&1&0&1&0&1&1&0&1&0&0&1\\
0&0&&0&0&1&1&1&1&0&0&0&0&1&0&0&1&1&0&0\\
0&0&&0&0&0&1&1&1&1&0&0&0&0&1&0&0&1&1&0\\
0&0&&1&0&0&0&1&1&1&1&0&0&0&0&1&0&0&1&1\\
0&0&\cdots&0&1&1&1&1&0&1&0&1&1&1&1&1&0&0&0&1
\end{bmatrix} \tag{45}$$

Daraus lässt sich die Prüfmatrix, wie in Anhang C.2 beschrieben ableiten

$$\mathbf{H}_{\mathrm{SEF}} = \begin{bmatrix}
1&0&0&0&1&1&1&1&0&0&0&0&1&0&0&1&1&0&\cdots&0&0\\
1&1&0&0&1&0&0&0&1&0&0&0&1&1&0&1&0&1&&0&0\\
1&1&1&0&1&0&1&1&0&1&0&0&1&1&1&1&0&0&&0&0\\
1&1&1&1&1&0&1&0&1&0&1&0&1&1&1&0&0&0&&0&0\\
0&1&1&1&1&1&0&1&0&1&0&1&0&1&1&1&0&0&&0&0\\
1&0&1&1&0&0&0&1&1&0&1&0&0&0&1&0&0&0&&0&0\\
0&1&0&1&1&0&0&0&1&1&0&1&0&0&0&1&0&0&&0&0\\
1&0&1&0&0&0&1&1&0&1&1&0&0&0&0&1&0&0&&0&0\\
1&1&0&1&1&1&1&0&1&0&1&1&1&0&0&1&0&0&&0&0\\
1&1&1&0&0&0&0&0&0&1&0&1&0&1&0&1&0&0&&0&0\\
1&1&1&1&1&1&1&1&0&0&1&0&0&0&1&1&0&0&&0&0\\
1&1&1&1&0&0&0&0&1&0&0&1&1&0&0&0&0&0&&0&0\\
0&1&1&1&1&0&0&0&0&1&0&0&1&1&0&0&0&0&&0&0\\
0&0&1&1&1&1&0&0&0&0&1&0&0&1&1&0&0&0&&1&0\\
0&0&0&1&1&1&1&0&0&0&0&1&0&0&1&1&0&0&\cdots&0&1
\end{bmatrix} \tag{46}$$

Multipliziert man einen Fehlervektor $\mathbf{E}$ mit $\mathbf{H}^{\mathrm{T}}$, so erhält man ein Syndrom $\mathbf{S}$. Es kann gezeigt werden, dass für alle 1-, 2- und 3-fach Fehler disjunkte Syndrome entstehen. Beschränkt man sich auf Fehler innerhalb der Informationsstellen der Codewörter, so existieren $\binom{16}{1} + \binom{16}{2} + \binom{16}{3} = 696$ Syndrome.

Das Aufstellen der Generator-Matrix, das Systematisieren, das Ableiten der Prüfmatrix, die Berechnung aller interessierenden Syndrome und die Prüfung auf deren Disjunktheit wurde mit Hilfe eines Programms in C++ im Zuge dieser Arbeit erledigt.

2.7.6.6. Reed-Solomon-Codes Die Gruppe der Reed-Solomon-Codes ist eine Teilmenge der nichtbinären BCH-Codes. Sie haben die Eigenschaft, dass die vorgegebene Minimaldistanz bei der Aufstellung des Codes gleich der tatsächlich erreichten Minimaldistanz ist. Sie sind damit Maximum-Distanz-Codes. Grundlage ist ein Feld GF(p^m), wobei p prim sein muss.

Blocklänge	$n = p^m - 1$
Anzahl der Prüfbits	$c = (n - k) = 2t_d$
Minimaldistanz	$d_{\text{min}} = 2t_d + 1$

Da Reed-Solomon-Codes nichtbinäre Codes sind, ist es nötig, mehrere Bits innerhalb des Datenwortes zu einem nichtbinären Codesymbol zusammen zu fassen.

Nimmt man GF(2^2) als Grundlage, so erhält man eine Blocklänge von 3, wobei jedes Codesymbol aus 2 Bits vom Datenwort zusammengesetzt wird, sodass dies effektiv einer Bitlänge von 6 Bits entspricht. Dies reicht nicht aus, um damit den RAM des MSP430 zu codieren.

Beim GF(2^3) erhält man eine Blocklänge von 7, wobei jedes Codesymbol aus 3 Bits vom Datenwort besteht. Damit ist die effektive Bitlänge 21. Will man aber einen systematischen Code aufbauen, so ist es nötig Daten- und Prüfbits voneinander zu trennen. Da 16 Datenbits mit 6 Codesymbolen im GF(2^3) dargestellt werden können, bleibt nur ein Codesymbol für die Prüfbits übrig. Damit wäre noch nicht einmal ein Einzelfehler korrigierbar.

Die nächste Möglichkeit, GF(2^4) bietet eine Blocklänge von 15, wobei jedes Codesymbol aus 4 Bits vom Datenwort zusammengesetzt ist. Dies bedeutet eine effektive Bitlänge von 60 (wobei 16 Bits für das Datenwort benötigt werden), was sich für den RAM des MSP430 als viel zu groß heraus stellt, um kostengünstig realisierbar zu sein.

Wünschenswert wäre daher ein Code, der z. B. den doppelten Bedarf an RAM hat, wie der ungeschützte RAM und dabei maximal viele Fehler korrigieren kann. Dieser kann gefunden werden, indem man wieder GF(2^2) als Grundlage nimmt, also jeweils 2 Datenbits zu einem Codesymbol zusammen fasst. 16 Datenbits eines Datenwortes müssen also mit 8 Codesymbolen repräsentiert werden. Spendiert man nun 7 Codesymbole als Prüfbits, so erhält man eine Blocklänge von 15, die der eines Reed-Solomon-Codes über GF(2^4) entspricht. Wünschenswert ist also die Existenz eines Codes mit Symbolen in GF(2^2), aber mit der Blocklänge und den daraus sich ergebenden Fehlerkorrektureigenschaften des GF(2^4). Reduziert man das Codesymbolalphabet des GF(2^4) auf das des GF(2^2), kann dies erreicht werden.

Sei also GF(2^4) die Ausgangsbasis. Sei α ein primitives Element in GF(2^4), so würde der entsprechende Reed-Solomon-Code das folgende Generatorpolynom besitzen:

$$g(x) = \prod_{i=1}^{7}(x - \alpha^i) \tag{47}$$

Bei der Berechnung dieses Produktes verwendet man nun aber nicht GF(2^4) und ein passendes primitives Generatorpolynom, sondern GF(2^2), hier im Beispiel generiert durch das Generatorpolynom $p(x) = x^2 + x + 1$. (Siehe dazu Anhang C.4.) Anschaulich gesagt, reduziert man damit das Codesymbolalphabet. Damit erhält man als Generatorpolynom für den Code das Polynom

$$g(x) = x^7 + \alpha x^6 + x + \alpha \tag{48}$$

mit Koeffizienten in GF(2^2).

Da die Berechnung etwas aufwändig ist, sollte man ein Mathematikprogramm dafür verwenden, welches symbolisch rechnen kann. Am Beispiel von Maple soll dies demonstriert werden:

Code	Erläuterung
`g:=product( x-a^y , y=1..7);`	Gleichung (48)
`gsort:=sort( collect( expand((g), x ) ,x) );`	Ausmultiplizieren, Sortieren nach Potenzen von x
`koeff:=Array(0..7);`	geeignetes Array
`gfeld:=GF(2,2,a^2+a+1):`	Definition von GF(2^2) mit $p(x) = x^2 + x + 1$
`for i from 2 by 1 to (7+1) do` `koeff[(7+2)-i]:=gfeld[ConvertIn](op(i,gsort)/x^((7+2)-i));` `end do;`	Extrahierden der Koeffizienten in a und Konvertierung in das GF(2^2)

Das Array `koeff` enthält dann die Koeffizienten des Generatorpolynoms für den zyklischen Code, siehe (48). Damit kann man dann die Generatormatrix des zyklischen Codes aufstellen.

$$\mathbf{G} = \begin{matrix} \alpha & 1 & 0 & 0 & 0 & 0 & \alpha & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \alpha & 1 & 0 & 0 & 0 & 0 & \alpha & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \alpha & 1 & 0 & 0 & 0 & 0 & \alpha & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \alpha & 1 & 0 & 0 & 0 & 0 & \alpha & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \alpha & 1 & 0 & 0 & 0 & 0 & \alpha & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \alpha & 1 & 0 & 0 & 0 & 0 & \alpha & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \alpha & 1 & 0 & 0 & 0 & 0 & \alpha & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \alpha & 1 & 0 & 0 & 0 & 0 & \alpha & 1 \end{matrix} \tag{49}$$

Diese Matrix kann man nun leider nicht in die standart echelon form bringen. Zur besseren Übersicht ist hier die Tabelle für die Addition und die Multiplikation in dem beschriebenen GF(2^2) angegeben in Abbildung 12. Damit ist ersichtlich, dass keine Ad-

$+$	0	α	1	α^2
0	0	α	1	α^2
α	α	0	α^2	1
1	1	α^2	0	α
α^2	α^2	1	α	0

Abbildung 12: Addition in GF(2^2), generiert durch $p(x) = x^2 + x + 1$

dition von zwei Zeilen oder von den daraus entstehenden Ergebnissen dazu führen kann, dass im linken 8×8 Block der Matrix (der der Informationsteil in der standart echelon form ist) nur die Hauptdiagonale besetzt ist.

Da kein systematischer linearer Blockcode aufgebaut werden kann, existiert keine Möglichkeit der einfachen parallelen Codierung und Decodierung. Eine Decodierung mittels look-up-table ist viel zu aufwändig zu realisieren.

Weiterhin erscheint schon der Aufwand für die look-up-table für die Fehlerkorrektur sehr hoch, da $3\binom{8}{1} + 3^2\binom{8}{2} + 3^3\binom{8}{3} = 1788$ mögliche Fehlervektoren existieren.

Die einzige Alternative wäre also ein sequentieller Decodieralgorithmus, wie in [19] vorgestellt. Dies ist aber nicht ohne Taktüberhöhung im MSP430 transparent implementierbar.

2.7.6.7. Schritte für die Implementierung im MSP430 In Kapitel 2.7.6.1 wurden schon Vorbetrachtungen für den RAM gemacht. Von den vorgestellten Codiermöglichkeiten erscheinen Parity (Kapitel 2.7.6.2), der (7,4)-Hamming-Code sowie der verkürze (15,11)-Hamming-Code, der ein (12,8)-Code ist (Kapitel 2.7.6.3) und der BCH-Code (Kapitel 2.7.6.5) geeignet für eine Implementation.

Bei der Realisierung von einem Parity-Check stellen sich keine großen Probleme. Die Codier- und Prüfschaltung sind einfach und transparent in den Datenpfad einzubinden. Aufgrund der Möglichkeit wahlfrei auf das low- oder high-Byte oder das gesamt Word zu schreiben, kann man einen Parity-Check für jedes RAM-Byte realisieren oder man muss bei einem Parity-Check über das gesamte word die Check-Bits in einem disjunkten Teil des RAM ablegen.

Ebenfalls einfach zu realisieren ist der (7,4)-Hamming-Code, da bei diesem die Daten-Bytes des RAM unabhängig voneinander betrachtet werden können. Für jedes Byte wird das entsprechende Codewort unabhängig vom anderen erzeugt und beim Lesen auf Fehler überprüft. Die Matrizen-Operationen, die für die Codierung und Syndrombestimmung notwendig sind, kann man leicht mit AND- und XOR-Gattern realisieren. Weiterhin ist eine look-up-table notwendig, um dem bestimmten Syndrom einen Fehlervektor zuzuordnen. Ist dieser gefunden, muss er zum empfangenen Datenwort dazu addiert werden (bitweise XOR). Dank des systematischen Aufbaus der Generator-Matrix ist eine Decodierung sehr simpel.

Für die Umsetzung des (15,11)-Hamming-Codes, der zu einem (12,8)-Code verkürzt wurde, sind jeweils 4 Bit pro data byte als Prüfbits nötig. Besonders simpel ist die Realisierung, wenn für jedes codierte data-byte exklusiv 12 Bit vorhanden sind. Verwendet man stattdessen für den RAM 3×8 Bit RAM-Blöcke, so müssen in einem RAM-Block die Check-Bits von beiden data-bytes abgelegt werden. Generell erscheint dieser Code die sinnvollste Alternative für eine Realisierung im MSP430 zu sein. RAM in gängigen Computerarchitekturen ist meist als „ECC-RAM“ mit einem ähnlichen verkürzen Hamming-Code versehen. ((288,256)-Code, der aus 4 (72,64)-Codes besteht, die auf dem (127,120)-Hamming-Code basieren, der um ein Parity-Bit auf ein (128,120)-Code erweitert wurde.)

Die vergleichsweise aufwändigste, aber auch leistungsstärkste geeignete Möglichkeit ist der BCH-Code.

Alle in diesem Kapitel genannten Möglichkeiten zeichnen sich dadurch aus, parallel zu arbeiten und somit durch rein kombinatorische Logik darstellbar zu sein. Sie können

damit völlig transparent in den Datenpfad des MSP430 implementiert werden und es wird für diese Komponenten auch keine Taktüberhöhung benötigt. Die einzige Auswirkung, die sich auf den Microcontroller nieder schlägt, ist die vergrößerte Signallaufzeit vom Prozessor zum RAM und zurück. Damit wird der maximal erreichbare Takt des MSP430 sinken.

Geht man davon aus, dass die eingesetzte Fehlerschutzkorrektur ausreichend ist, so ist nichts weiter zu tun. Will man aber signalisieren, dass ein Fehler aufgetreten ist, um so anzuzeigen, dass etwas nicht in Ordnung ist, so muss eine Komponente in den MSP430 implementiert werden, die einen Interrupt auslöst, der eine geeignete Programmroutine startet. In Kapitel 2.9.3 wird darauf noch näher eingegangen.

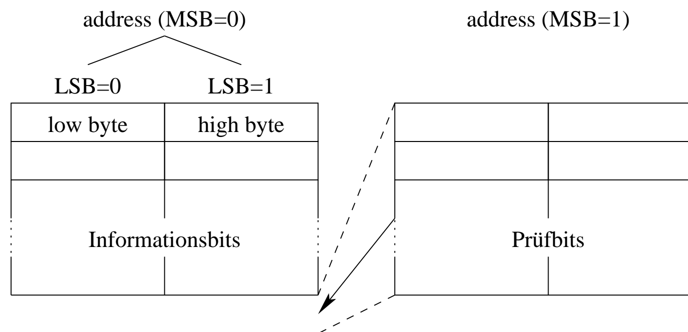


Abbildung 13: ECC- oder Data RAM

Der BCH-Code für den RAM basiert auf der Verwendung der (nahezu) doppelten Menge an RAM. Dieser Einsatz erscheint recht kostspielig. Es würde sich daher anbieten, die Fehlerschutzcodierung auf Basis eines BCH-Codes im Bedarfsfall abschaltbar zu machen und den RAM für den Fehlerschutz stattdessen als Daten-RAM zu verwenden.

Da der RAM des MSP430 aus zwei 8 Bit-RAM-Blöcken besteht [8], ist es nötig, den zusätzlichen RAM für die Fehlerschutzcodierung ebenso auszulegen, wenn man ihn auch als Daten-RAM verwenden will. Da beim Schreibzugriff auf ein Byte mit angeschalteter Fehlerschutzcodierung wie oben erwähnt das jeweils andere Byte gelesen werden muss, um die Prüfbits zu berechnen, muss der zusätzliche RAM disjunkt zum normalen RAM sein. Es ist nicht möglich, z. B. einen 32 Bit-RAM für ein volles fehlerschutzcodiertes Datenwort zu verwenden.

Damit kann also teurer Dual-Port-RAM verwendet werden, oder man muss zwei weitere 8 Bit-RAM-Blöcke für die Prüfbits, also insgesamt 4 mal 8 Bit-RAM-Blöcke verwenden. Abbildung 13 zeigt die Adressierung in Verbindung mit dem zusätzlichen RAM, wenn dieser für die Prüfbits oder als normaler Daten-RAM verwendet wird.

Beim Einsatz des (12,8)-Hamming-basierten-Codes ist ein ähnliches Vorgehen nicht möglich, da für high- und low-byte jeweils disjunkte RAM-Blöcke verwendet werden müssen und so nicht einfach die gerade und ungerade Adresse dieses dritten RAM-Blockes für die beiden Data-Bytes verwendet werden kann. Lediglich beim Verwenden von Dual-Port-RAM wäre der Einsatz als Data-RAM denkbar.

2.7.7. RNS

RNS (residue number system) ist die Bezeichnung für Zahlensysteme, mit denen man Datenwörter in einem nicht-binären System, basierend auf einer Basis aus Zahlen, die paarweise relativ prim sind, darstellt. Es ist bekannt, dass jede ganze Zahl $k \in \mathbf{Z}$ durch das Produkt aus Primzahlen darstellbar ist.

Sei nun $(m_{n-1}, m_{n-2}, \ldots, m_1, m_0)$ ein n-Tupel von Zahlen $m \in \mathbf{Z}$, die paarweise relativ prim sind. Dieses n-Tupel bildet eine Basis mit der alle Zahlen $k \in [0, M-1]$, mit $M = \prod_{i=0}^{n-1} m_i$ eindeutig darstellbar sind. Die Darstellung erfolgt komponentenweise, sodass $k = (k_{n-1}, \ldots, k_1, k_0)$ mit $k_i \in \{0, 1, \ldots, m_i - 1\}$. Jede Komponente k_i wird also berechnet mit $k \bmod m_i$ und formt damit ein Residuum. Den Beweis für die Eindeutigkeit liefert das „Chinese Remainder Theorem", wie es z. B. in [20], [21] oder auch in [22] vorgestellt wird.

Ein RNS besteht somit in seinen Komponenten aus Ringen und formt selber einen Ring. Damit sind eine Addition und eine Multiplikation (modulo m_i für die Komponenten bzw. modulo M) definiert.

Es soll im folgenden an einem Beispiel die Erläuterung fortgesetzt werden. Das Beispiel stammt in seiner Grundform aus [23] und wurde passend erweitert.

Sei $(m_2, m_1, m_0) = (5, 3, 2)$. Damit ist $M = 30$.

k	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	29	30	31
k_0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	...	1	0	1
k_1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	...	2	0	1
k_2	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	...	4	0	1

Eine Zahl $k = 5$ wird demnach wie folgt in diesem RNS dargestellt:

$$|5|_{30} = (|5|_5, |5|_3, |5|_2) = (0, 2, 1) \tag{50}$$

Addition und Multiplikation laufen komponentenweise ab:

$$\begin{aligned} |4+5|_{30} &= (4,1,0) + (0,2,1) = (4,0,1) = |9|_{30} \\ |3 \times 5|_{30} &= (3,0,1) \times (0,2,1) = (0,0,1) = |15|_{30} \end{aligned} \tag{51}$$

Damit ist auch der größte Vorteil von RNS gezeigt: Addition und Multiplikation sind „Carry-frei". Die Geschwindigkeit wird also allein durch die Rechnung mit dem „langsamsten Modulus" bestimmt.

Wie leicht zu erkennen ist, enthält dieses RNS noch keine Redundanz, um Fehler zu erkennen oder zu korrigieren. In [24] wird die Aussage getroffen, dass ein Fehler erkennbar ist, wenn $n+1$ Residuen-Kanäle für die Rechnung verwendet werden, wobei alle Kombinationen aus n Residuen-Kanälen die beabsichtigte Rechnung durchführen können müssen. Um beim obigen Beispiel zu bleiben, kann man leicht sehen, dass das RNS, aufgebaut aus $(m_1, m_0) = (3, 2)$ das Teilsystem mit dem kleinsten Wertebereich beschreibt. Bleibt man also mit jeder Rechnung mit dem gesamten System innerhalb dieses „kleinsten Teilsystems", so kann jeder Fehler in einem Residuen-Kanal detektiert werden. Anschaulich formuliert kann also im Wertebereich $k \in [0, 5]$ je einer der 3 Residuen-Kanäle ausfallen (einen fehlerhaften Wert liefern) und dies wird erkannt.

Das heißt: Liegt das Ergebnis der Rechnung außerhalb des Wertebereichs des „kleinsten Teilsystems“, muss ein Fehler aufgetreten sein.

Kurz: Man kann einen Fehler in einem einzelnen beliebigen Residuen-Kanal erkennen, wenn man einen Kanal mit $m_n > m_i$ für alle $m_i \in [0, n-1]$ dem RNS hinzufügt - oder den Wertebereich des RNS auf den des „kleinsten Teilsystems“ aus $n-1$ Komponenten beschränkt. Im obigen Beispiel ist der Gewinn durch diese Redundanzzugabe noch gering, da 3 Kanäle verwendet werden anstatt 2, aber wenn man ein RNS, bestehend aus mehr Residuen-Kanälen verwendet, steigt die Effizienz dieser Methode an.

Offensichtlich lassen sich Addition und Multiplikation in einem RNS gut darstellen. Problematisch ist allerdings Overflow-Detektion. Andere Operationen, wie sie ein Microcontroller, wie der MSP430 bietet (AND, XOR, ...) sind dagegen gar nicht darstellbar. Die Idee, den gesamten Datenpfad des Microcontrollers mit Residuen-Arithmetik aufzubauen, muss also verworfen werden.

Eventuell könnte also noch der Adder des MSP430 mit Residuen-Arithmetik aufgebaut werden. Hier stellt sich also das Problem, dass die normalen Binärdaten in ein RNS und zurück gewandelt werden müssen. [25], Seite 11 zeigt einen Ansatz über vorberechnete Residuen. Der Ansatz kann einfach in sequentieller Logik und mit einigem Aufwand in kombinatorischer Logik realisiert werden. Eine Rückwandlung könnte prinzipiell mit Hilfe des Chinese Remainder Theorems erfolgen:

$$k = \left[\sum_{i=0}^{n-1} M_i \left| k_i L_i \right|_{m_i} \right] \bmod M \tag{52}$$

Dabei sind $M_i = \frac{M}{m_i}$ und L_i die multiplikative Inverse innerhalb des Ringes i, also $|M_i L_i|_{m_i} = 1$

Es ist also zu sehen, dass die Wandlung in ein RNS und zurück einen beträchtlichen Rechenaufwand mit sich bringt. Dieser Aufwand wird sich nur auszahlen, wenn z. B. schnellen Multiplikationen mit hohen Bitbreiten realisiert werden müssen, wie es in DSPs häufig der Fall ist. Außerdem wird eine effiziente Realisierung der Hin und Rückkonvertierung meist sequentiell ablaufen. Für eine Implementierung in einem low-power und low-cost Microcontroller, wie dem MSP430 erscheint ein RNS nicht geeignet.

2.7.8. Sicherheit der state machine

Oft wird bei der Realisierung von digitalen Schaltungen auf state machines zurück gegriffen, so auch in [8]. Da die dort implementierte state machine von zentraler Bedeutung für den gesamten Microcontroller ist, würde sich ein Fehler innerhalb der Logik für diese state machine fatal auswirken. Es wäre daher sinnvoll, eine Absicherung des state machine zu realisieren.

2.7.8.1. Die state machine als Linear Digital State Variable System [18] zeigt eine Möglichkeit, Fehler innerhalb Linear Digital State Variable Systems zu erkennen.

Ausgehend von

$$\begin{aligned} s_{t+1} &= As_t + Bx_t \\ y_t &= Cs_t + Dx_t \end{aligned} \tag{53}$$

wenn s der Zustand des Systems, x der Eingabevektor, y der Ausgabevektor, t die Zeit (als Taktschritt) und A, B, C, D die Übertragungsmatrizen sind, werden in dieser Arbeit zwei Testmöglichkeiten hergeleitet, die prinzipiell auch auf state machines, wie sie üblicherweise in in einer Hardwarebeschreibungssprache modelliert werden, anwendbar sind.

Das große Problem in der Praxis ist die große Anzahl von states. Dabei sind nicht nur die states der state machine in die Kalkulation einzubeziehen, sondern auch sämtliche anderen Speicherelemente, wie Register und RAM, da der Ablauf der state machine unter anderem auch von diesen Speicherinhalten abhängt. Damit werden die Vektoren und Matrizen aus (53) außerordentlich groß, sodass praktisch keine Berechnung derselben mehr möglich ist.

Selbst unter der Bedingung, dass weder RAM noch Register Einfluss auf eine state machine hätten, gestaltet sich der Ansatz für den Schaltkreisdesigner außerordentlich aufwändig. Im Realfall wird in einer Hardwarebeschreibungssprache eine state machine als high-level - Verhaltensbeschreibung entwickelt, sodass die Vektoren und Matrizen aus (53) nicht direkt ablesbar sind. Um sie zu erhalten, müssten sie manuell aus dem Code der Hardwarebeschreibungssprache extrahiert werden. Sollten sich im Lauf des Entwicklungsprozesses Änderungen an der state machine ergeben, so sind die Matrizen aus (53) wieder manuell zu ändern. Dieser Ansatz erscheint somit nicht praktikabel.

2.7.8.2. Eine Ablaufkontrolle für state machines Unter der Bedingung, dass der Zustandsgraphen der state machine analog wie in Abbildung 14 aussieht, ist eine sehr einfach zu realisierende Version einer Ablaufkontrolle in Form eines „state-Zählers“ denkbar, wenn zusätzlich folgende das problem vereinfachende Bedingungen gelten:

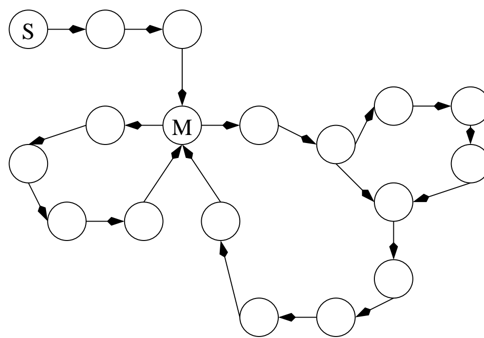


Abbildung 14: Eine state machine

1. Jede Schleife oder Verzweigung im Zustandsgraphen muss in einer eindeutigen Anzahl von Taktschritten durchlaufen werden.

2. Es gibt einen master-state M, an dem die Verzweigungen starten.

3. Schleifen innerhalb von Schleifen sind erlaubt, wenn beim Einsprung in die äußere Schleife mit der Einsprungbedingung klar ist, ob die innere Schleife durchlaufen wird.

Die in [8] realisierte state machine genügt diesen Voraussetzungen. Der master-state ist der Zustand des Prozessors, in dem ein neuer Befehl aus dem RAM geholt und decodiert wird. Die einzelnen Schleifen repräsentieren die Zustände, die zur Abarbeitung eines Befehls nötig sind.

Einen „state-Zähler" könnte man nun auf verschiedene Weise realisieren. Die einfachste Möglichkeit besteht darin, jeden durchlaufenen state zu zählen und im master-state den Zähler auszuwerten und danach rückzusetzen. Da man durch den Befehl genau weiß, wieviele states nötig sind, um diesen abzuarbeiten, kann man das Zählergebnis mit der erwarteten Anzahl der durchlaufenen states vergleichen. Damit ist es möglich zu detektieren, ob die state machine eine Schleife durchlaufen hat, die genau so lang ist, wie für den entsprechenden Befehl erwartet. Existieren mehrere Schleifen mit der selben state-Anzahl, so ist nicht detektierbar, ob die richtige Schleife durchlaufen wurde.

Dieses Zähler-Prinzip kann man verfeinern. Zählt man nicht einfach jeden beliebigen state, sondern zählt man nur genau dann, wenn die state-Variable eine mögliche Codierung trägt, die zu dem abzuarbeitenden Befehl passt, so kann man ausschließen, dass eine falsche Schleife angesprungen wurde.

Möchte man nun noch ausschließen, dass der Fall eintritt, dass zwar die korrekte Schleife angesprungen, aber innerhalb der Schleife die Reihenfolge der states nicht korrekt abgearbeitet wurde, dennoch aber zufällig nach genau der erwarteten Anzahl von Taktzyklen wieder der master-state erreicht wurde, so darf man den Zähler nur dann erhöhen, wenn der aktuelle state eine Codierung trägt, die zu dem abzuarbeitenden Befehl passt und zusätzlich der vorherige state tatsächlich dessen erlaubter Vorgänger war. Der Hardwareaufwand für diese erweiterte Schutzfunktion wird allerdings sehr groß sein.

Äste, die keine Schleifen repräsentieren, wie in Abbildung 14 vom start-state S zum master-state M stellen beim Zählen kein Problem dar. Die einzige Modifikation ist die, den Zähler auch im state S rückzusetzen und im state M geeignet die Zähleranzahl zu überprüfen.

Wie erwähnt sind die oben erwähnten Bedingungen nur nötig, um einige Vereinfachungen zu erreichen. Ist z. B. Bedingung 3 verletzt, so können prinzipiell weitere master-states an den Verzweigungspunkten hinzugefügt werden, was das Problem etwas komplizierter macht. In diesem Fall ist muss in jedem master-state der Zähler zurückgesetzt und beim Erreichen eines anderen master-states die Anzahl korrekt ausgewertet werden. Dies bringt je nach Aufbau der state-machine etwas zusätzliche Logik, sowie die Notwendigkeit mit sich, zusätzlich zum Befehl (der Verzweigungsbedingung) noch den alten master-state abzuspeichern.

Sollte einmal die gezählte state-Anzahl nicht mit der erwarteten übereinstimmen, wäre es sehr sinnvoll, wenn ein Interrupt generiert werden könnte, sodass angepasste

Software darauf reagieren könnte. Sollte der master-state im Fehlerfall nie wieder erreicht werden, so greift diese Prüfung auf die Anzahl der states und die Generierung eines Interrupts ins Leere. Dies kann aber leicht von einem Watchdog (Kapitel 2.7.5) abgefangen werden.

Verfolgt man dieses Problem weiter, wäre sogar ein „state-Watchdog“ denkbar: Man könnte den state-Zähl- und Test-Schaltkreis wie folgt modifizieren: Wenn eine bestimmten state-Anzahl (die größer ist als die Anzahl der states in der längsten Schleife des Zustandsgraphen) erreicht wird oder die Zähl-Variable sich nicht mehr ändert und der master-state aber inzwischen hätte erreicht werden müssen, dann kann man einen Reset auslösen.

Mit der hier vorgestellten Idee einer Ablaufkontrolle ist es möglich, auszuschließen, dass ein falscher Befehl abgearbeitet wird. Dafür ist es nötig, dass gewährleistet ist, dass der Befehl im RAM nicht verfälscht und korrekt in die CPU in das Befehlsregister geladen wurde. Speziell Fehler beim state-Wechsel, die in Sprüngen zu falschen Punkten im Zustandsgraph führen, sind detektierbar.

2.7.9. Eigentest

Gängige Praxis ist es, digitale Schaltungen mit Möglichkeiten zum Test auszustatten. Eine der bekanntesten Alternativen ist ein Scan-Path. Es liegt nahe, diese Möglichkeiten nicht nur für den einmaligen Test nach der Fertigung, sondern auch während des Betriebes zu nutzen. Meist wird dies nur als Offline-Test möglich sein, aber dennoch erscheint es attraktiv.

Am Beispiel von Scan-Path sollen einige grundlegende Überlegungen gemacht werden. Da der Scan-Path-Test ein Offline-Test ist, kann er nur bei einem POST (power-on self test) oder während „Arbeitspausen“ durchgeführt werden. Generell muss beachtet werden, dass dieser Test relativ kurz gehalten wird. Ein langwieriges Booten oder eine lange Offline-Phase ist meist nicht akzeptabel.

Die Realisierung als POST erscheint die einfachste Möglichkeit für eine Implementierung zu sein. Man kann im Prinzip den kompletten normalen Funktionstest, wie er auch nach der Fertigung gemacht wird, einsetzen. Einzig die Größe eines ROM, der die Testvektoren und die jeweils erwarteten Ergebnisse aufnimmt, ist kritisch.

Möchte man in „Arbeitspausen“ einen Test durchführen (z. B. weil der Prozessor kontinuierlich arbeitet, also sehr selten abgeschaltet und neu gestartet wird), so ist es sehr wichtig, zuerst den aktuellen Zustand des Prozessors zu sichern. Dazu gehören alle Register (also inklusive PC, SP und SR). Dann muss gewährleistet sein, dass bei dem Test nicht auf einen RAM-Bereich schreibend zugegriffen wird, der auch vom Programm während dieser Pausenzeit als Programm- oder Arbeitsspeicher genutzt wird. Im allgemeinen kann der Test nicht abgebrochen werden, der Prozessor ist also nicht ansprechbar. Will man einen Testabbruch aber dennoch realisieren, so ist eine geeignete Interrupt-Routine in den Test-Chip zu integrieren. Diese muss den Test stoppen und den Prozessor in den Ursprungszustand versetzen - ebenso, wie nach einem normalen Test-Ende.

Problematisch ist es natürlich, eine „Arbeitspause“ als solche zu erkennen. Da ein Scan-Path-Test im allgemeinen sehr lange dauert, muss gewährleistet werden können, dass der Prozessor mindestens diese Zeit lang definitiv nicht benötigt wird. Das ist ein anwendungsabhängiges Problem und wird sich oft nur durch höherwertige Intelligenz lösen lassen, indem z. B. ein Host-Rechner diesen Pausen-Zustand meldet.

Bis jetzt sind im MSP430 nach [8] noch keine derartigen Teststrukturen wie Scan-Path implementiert, die genutzt werden könnten.

2.7.10. FPGA

Der Einsatz von Field Programmable Gate Arrays (FPGA) bietet einige Vorzüge bei dem Aufbau eines fehlertoleranten Systems. So besteht aufgrund der regelmäßigen Struktur die Möglichkeit, mittels ladbarer Testkonfigurationen jeden einzelnen Block der FPGA auf Fehler zu überprüfen [16]. Dieser Test, der als Offline-Test läuft, kann somit stets bei Bedarf ausgeführt werden, so z. B. in einer größeren Pause für das System oder als POST. In Kapitel 2.7.9 wurde ein ähnliches Prinzip schon beschrieben.

Des weiteren bieten FPGAs die Möglichkeit, nach dem Entdecken eines Fehlers darauf schnell und äußerst flexibel reagieren zu können. Ist ein Block der FPGA fehlerhaft, so kann dieser z. B. aus dem Routing-Prozess ausgeschlossen werden, wenn man das Design einem neuen Routing & Mapping für die FPGA unterzieht [9], [10]. Selbstverständlich ist dies in der Form nur von einem menschlichen Administrator machbar. Da aber Offline-Zeiten, die durch das neue Routing & Mapping anfallen, in der Praxis meist inakzeptabel groß sind, besteht die Möglichkeit, alternative Konfigurationen mit dem System mitzuführen und bei Bedarf automatisch zu laden.

Der Einsatz FPGAs ist vor allem dann zu empfehlen, wenn über lange Zeit kein Techniker an das System heran kann, um einen Fehler zu beheben. Die Raumfahrt, wo nur eine Fernüberwachung über einen Datenkanal mit sehr begrenzter Bandbreite zur Verfügung steht, ist ein typisches Anwendungsgebiet. Da aber FPGAs immer kostengünstiger werden, ist ihr Einsatz auch in Produkten für den Massenmarkt denkbar. Gerade in der Automobilbranche kann es effektiv sein, Reparatur-Montagen, die durch den Austausch von defekten Schaltkreisen entstehen, einzusparen und stattdessen nur eine andere FPGA-Konfiguration zu laden.

Nicht zu unterschätzen ist auch die Möglichkeit, bestimmte Signalverarbeitungsaufgaben nach der Auslieferung des Automobils zu verändern. Trotz des hohen Aufwands, ein fehlerfreies Produkt auf den Markt zu bringen, ist es nicht ausgeschlossen, dass Fehler erst nach der Auslieferung entdeckt werden.

2.8. Möglichkeiten des Bussystems

Ist ein digitaler Bus am Ausgang des intelligenten Sensor-Systems angeschlossen, wie beispielsweise I^2C oder CAN, so kann ein beliebiges Protokoll definiert werden, das definiert, wie Daten und Fehler signalisiert werden. Bidirektionaler Datentransfer ist ebenfalls möglich. Somit stellt ein digitaler Bus ein hohes Maß an Flexibilität dar.

Schwieriger ist es, wenn ein analoger Bus verwendet wird, wie ein „current loop“

(siehe dazu [26]) oder am Ausgang des Sensorsystems einfach eine Spannung als Signal bereit steht. Derart analoge Bussysteme bieten die Möglichkeit, über große Entfernungen eine einfache Signalübertragung realisierbar zu machen. Weiterhin kann auf Grund des einfachen Aufbaus auch ein „dummes Anzeigegerät" an diesen Bus angeschlossen werden. Solche Geräte sind analoge Zeigerinstrumente oder simple Digitalanzeigen, die entweder Strom oder Spannung direkt messen und auf ihrer Anzeige einen entsprechenden Wert ausgeben.

Sei im Beispiel nun ein „current loop" gegeben, in dem 4 bis 20 mA fließen dürfen. Als Anzeigeinstrumente werden ein Zeigerinstrument und eine Digitalanzeige, welche beide Strommessgeräte sind, betrachtet. Die Schwierigkeit ist nun, auf diesen Anzeigen in geeigneter Weise einen Fehler im Sensorsystem zu signalisieren, damit ein Überwacher den Fehler entdecken kann.

Die Digitalanzeige ist zwar frei einstellbar, welchen Wert das Display bei welchem Stromfluss anzeigen soll, aber das Problem ist die Messgenauigkeit. Einerseits muss der Ausgang des intelligenten Sensorsystems einen stabilen Strom bereit stellen und andererseits muss die Strommessung in der Anzeige mit entsprechender Genauigkeit arbeiten. Ist dies gegeben, so kann ein gewisser Bereich der Stromstärke definiert werden, in dem die Digitalanzeige einen bestimmten Fehlercode präsentiert. Umso mehr mögliche Fehler so signalisiert werden sollen, desto kleiner wird der Bereich für das normale Messsignal und damit die Anzeigegenauigkeit.

Bei einem analogen Zeigerinstrument können prinzipiell ebenso gewisse Bereiche der Stromstärke zu einem Zeigerausschlag führen, der in einem „verbotenen Bereich" auf der Anzeige liegt. Umso schmaler allerdings der Bereich für einen bestimmten Fehlercode gewählt wird (und mehrere Fehler unterschiedlich signalisiert werden sollen), desto schwieriger wird das Ablesen durch einen Überwacher. Weiterhin bietet sich aber auch die Möglichkeit an, dringliche Fehler speziell zu signalisieren, indem der Zeigerausschlag ständig zwischen Minimum und Maximum pendelt, sodass ein Überwacher dies sofort registrieren kann. Beim Kalibrieren des Gesamtsystems ist dabei selbstverständlich darauf zu achten, welche maximale Frequenz der Zeiger des Instrumentes noch darstellen kann. Sollte das Sensorsystem mit verminderter Genauigkeit aufgrund eines Fehlers weiterarbeiten, so kann dies ebenfalls angezeigt werden, indem der Zeiger um den Messwert herum pendelt und dabei den Toleranzbereich überstreicht, sofern dieser vom intelligenten Sensorsystem angegeben werden kann.

Generell sollte untersucht werden, ob für die Signalisierung die Möglichkeit besteht, Stromstärken außerhalb des definierten normalen Arbeitsbereiches (4 bis 20 mA) zu nutzen.

2.9. Erweiterungen für den Prozessor

Im folgenden sollen mögliche Erweiterungen für den Microcontroller MSP430 vorgestellt werden. Diese Erweiterungen dienen zur besseren Überwachung des intelligenten Sensorsystems und zur einfacheren Testbarkeit. Prinzipiell sind die hier vorgestellten Ideen auch auf anderen ähnlichen Prozessorarchitekturen realisierbar.

In [8] ist beschrieben, wie externe Komponenten an den MSP430 angeschlossen wer-

den. Die genaue Spezifikation dafür liefert [7]. Kurz: Externe Komponenten liegen in einem dafür reservierten RAM-Bereich des MSP430. Die RAM-Ansteuerung muss an diese Komponenten angepasst werden. Einige externe Komponenten können einen Interrupt auslösen. Dieser ist maskierbar über die SFR.

Soll eine neue Komponente die Möglichkeit besitzen, einen Interrupt auszulösen, so muss geprüft werden, welcher dafür zur Verfügung steht. Je nach Ausbau des Microcontrollers MSP430 existieren freie Interrupts, die dafür genutzt werden könnten. Eine ISR (Software) muss dann geeignet reagieren. Selbstverständlich bedeutet dies ein modifiziertes Design, da beim MSP430 theoretisch alle Interrupts für mögliche Komponenten von Texas Instruments reserviert sind. Im Realfall werden aber eben nicht alle dieser möglichen Komponenten angeschlossen sein - siehe die Modellpalette von Texas Instruments und [8].

Hat man nicht den MSP430 als Prozessor und kann man in diesem anderen Prozessor keinen IRQ reservieren, muss aber eine externe Komponente dennoch dem Prozessor ein Signal geben können, so sind die Möglichkeiten, die in Kapitel 2.7.4.15 beschreiben worden sind, auf ihre Machbarkeit hin zu untersuchen.

Generell muss eine genaue Spezifikation der neuen Hardwarekomponente existieren, damit der Software-Programmierer sie geeignet verwenden kann. Da beim MSP430 Interrupts genutzt werden müssen, die eigentlich von TI reserviert sind, muss bei der Software-Entwicklung darauf geachtet werden.

2.9.1. Ein nicht-flüchtiger Speicher

Ein nicht-flüchtiger Speicher bietet die Möglichkeit, z. B. eine Art Logbuch zu führen. Damit wäre es in einem intelligenten Sensorsystem möglich, Fehlerfälle zu protokollieren. Auch kann abgespeichert werden, welche (redundant ausgelegte) Komponente z. B. nach einem Reset und Neustart deaktiviert bleiben muss, da sie schon früher als fehlerhaft erkannt worden ist. Eine ganz andere Möglichkeit wird durch solch einem Speicher dem Benutzer geboten, der in diesem Speicher z. B. Konfigurationsdaten ablegen kann.

[7], Anhang C beschreibt ein EEPROM-Programming-Module. Dieses, bzw. diese Spezifikation könnte man verwenden, wenn man einen EEPROM als nicht-flüchtigen Speicher einsetzen möchte. Will man dagegen z. B. Flash-RAM einsetzen, so muss man eine geeignete Schreib-Komponente selber entwickeln. Bei beiden Ideen wird davon ausgegangen, dass der nicht-flüchtige Speicher im normalen Adressbereich des MSP430 liegt. Dies bietet sich vor allem dann an, wenn dieser nicht-flüchtige Speicher für Konfigurationsdaten genutzt werden soll. Problematisch erscheint aber der Fakt, dass damit ein Logbuch in diesem Speicher nicht besonders geschützt ist - es könnte im Fehlerfall überschrieben werden.

Man kann daher ohne Probleme auch eine Schreib-/Lesekomponente entwickeln, die einen disjunkten RAM ansteuert und auf den nur über diese Komponente zugegriffen werden kann. In Hardware kann dann gesichert sei, dass nie durch Softwarezugriffe auf eine Adresse schreibend zugegriffen werden darf, die für das Logbuch reserviert ist. Alternativ kann der Zugriff mittels Passwort ablaufen, sodass die Software stets ein Passwort zusätzlich zu den Daten senden muss, was die Wahrscheinlichkeit minimiert,

dass versehentlich auf eine falsche Adresse zugegriffen wird. (Siehe dazu beispielhaft die Realisierung des Watchdog im MSP430. [7]) Ein weiterer Vorteil wäre es, auf diesem Wege z. B. einen seriellen EEPROM anschließen zu können und damit kein paralleles Interface implementieren zu müssen.

Ist ein Lesezugriff auf diesen nicht-flüchtigen Speicher nötig und hat man nur einen langsamen (seriellen) Zugriff, so muss ein geeignetes Protokoll erstellt werden, um die Daten zuverlässig lesen zu können. Eine Signalisierung per Interrupt, wenn ein neues Datenwort aus dem nicht-flüchtigen Speicher gelesen wurde und im Adressbereich des Logbuches im Microcontroller vorzufinden ist, ist eine denkbare Möglichkeit. Eine andere Möglichkeit wäre es, bei einer bestimmten Adresse im RAM die Information abzulegen, ob ein neues Datum bereit steht. Ein Software-Algorithmus kann dies erkennen und das benötigte neue Datum abholen. Generell wird aber ein solcher Lesezugriff auf einen seriell angeschlossenen Speicher eher langsam ablaufen, sodass er nur während des POST ablaufen sollte.

Bei Schreibzugriffen ist selbstverständlich ebenfalls klar zu definieren, wieviele Takte die Ansteuerkomponente für den nicht-flüchtigen RAM benötigt, bis ein neues Datenwort angenommen werden kann.

Auf jeden Fall wird eine externe Erweiterungskomponente für den MSP430 benötigt. Diese kann gemäß der Spezifikation im Adressbereich des Microcontrollers abgelegt werden. Die 8×16 Bit RAM, die so im einfachsten Fall zur Verfügung stehen, sind ausreichend, um auch eine komplexe Kommunikation zwischen der Komponente und dem MSP430 möglich zu machen. Das Kommunikationsprotokoll des Multiplizierers, der im MSP430 integriert werden kann, bietet ein anschauliches Beispiel, wie solcherart Datenaustausch ablaufen könnte. ([7], [8])

Die Größe des nicht-flüchtigen RAM muss im konkreten Anwendungsfall abgeschätzt werden. Muss ein umfangreiches Logbuch angelegt werden und ist der intelligente Sensor lange Zeit ohne Wartung, so muss der Speicher dementsprechend groß dimensioniert werden. Will man nur Informationen ablegen, ob und wenn ja, welcher Sensor von mehreren Sensoren als defekt detektiert wurde, genügen ein paar Bits in diesem RAM. Umso größer der nicht-flüchtige RAM sein muss, desto eher muss man eine Realisierung in einem disjunkten RAM bevorzugen, da sonst der normale Adressbereich des MSP430 nicht ausreicht.

2.9.2. Ein Fehler-Logbuch

Im folgenden sollen einige Ansätze und Ideen vorgestellt werden. Da hier nur eine allgemeine Betrachtung erfolgen soll, werden die Ideen nur grob skizziert. Im konkreten Anwendungsfall ergeben sich sowieso meist automatisch die Anforderungen an ein derartiges Logbuch.

In Kapitel 2.9.1 wurde schon die Grundlage für ein Fehler-Logbuch vorgestellt: Ein nicht-flüchtiger RAM. Die Größe des RAM und dessen Anbindung (seriell / parallel) bestimmen den Umfang der zu speichernden Daten und die Geschwindigkeit, mit der die Daten abgelegt werden können.

2.9.2.1. Erstellung des Logbuches Oft reicht es nicht aus, nur per Software Fehlerfälle zu protokollieren, was den einfachsten Fall darstellt, wie er in Kapitel 2.9.1 schon beschreiben ist. Problematisch ist ein Logging in Software, wenn es zu einem Programmabsturz kommen würde, sodass der Watchdog - Timer einen Overflow produziert. Wenn nun nach dem anschließenden Reset wieder ein Fehler auftritt, kann eine Software-Routine möglicherweise nicht mehr beim POST nachprüfen, ob der Reset tatsächlich durch einen Watchdog-Overflow ausgelöst wurde. Auch wurden im ganzen Kapitel 2 mehrere Hardware-Lösungen vorgestellt, die einen Fehler signalisieren können. Diese Signalisierung könnte mit der hier vorgestellten Komponente für einen nicht-flüchtigen RAM verbunden werden, sodass automatisch durch eine Hardware-Routine das Fehlerprotokoll erweitert wird. Damit würde ein Fehler nur dann nicht protokolliert werden, wenn die Betriebsspannung ausfällt, oder ein vorangegangener anderer Fehler noch nicht fertig protokolliert wurde.

Es ist daher zweckmäßig nicht nur eine einfache I/O-Komponente zu einem nicht-flüchtigen RAM aufzubauen, sondern innerhalb dieser Komponente Routinen hinzuzufügen, damit ein automatisches Fehler-Logging in Hardware möglich ist.

2.9.2.2. Inhalt des Logbuches Zu untersuchen ist im Einzelfall immer, welche Informationen wirklich zwingend notwendig sind. Beispielsweise könnte der Fehlerfall der Überlast am Sensor nur einmal an eine bestimmte Adresse des Logbuches und nicht fortlaufend abgespeichert werden. Mit einem solchen Vorgehen für derartige Fehlerfälle kann man den RAM-Bedarf stark einschränken. Nicht sinnvoll ist dies aber, wenn transiente Fehler auftreten. Eine beispielsweise durch Parity-Check als fehlerhaft detektierte Rechnung oder das Auslösen eines Watchdog-Overflows sollte immer wieder als neuer zusätzlicher Logbuch-Eintrag protokolliert werden, um später erkennen zu können, ob der Fehler ein Einzelfall war, oder gehäuft aufgetreten ist.

Damit geht einher, dass stehts der „Füllstand“ des Logbuches bekannt sein muss. Dieser kann in dem nicht-flüchtigen RAM an einer festen Adresse gespeichert sein, sodass bei jedem Neustart des Systems immer der aktuelle Wert von der Logbuch-Komponente ausgelesen werden kann.

Im konkreten Einzelfall ist eine geeignete Codierung für die verschiedenen Fehlerfälle zu erstellen, sodass jedem möglichen Fehler ein eindeutiger Code zugeordnet werden kann.

Es kann sinnvoll erscheinen, auch eine Fehlerzeit dem Protokoll hinzuzufügen, um die Häufigkeit von Fehlern analysieren zu können. Diese Zeit kann als Zählung des angemessen geteilten Systemtaktes implementiert werden. Selbstverständlich bringt dies das Problem mit sich, dass ein Zählerüberlauf eintreten könnte, also muss die Taktteilung sinnvoll gewählt werden. Prinzipiell wäre es zwar auch möglich, nur die untersten n Bits des Zählers als Hardware-Zähler aufzubauen, die obersten m Bits aber in dem nicht-flüchtigen Speicher (gegebenenfalls über mehrere Datenwörter verteilt) abzulegen, um so einen Overflow zu verhindern, aber dies hat einen gravierenden Nachteil: Diese präzise Zeitangabe bringt in den meisten Fällen eine zu große Datenmenge mit sich, um sie in dem Logbuch sinnvoll abzuspeichern.

Daher muss für eine zu betrachtende Zeitspanne und für eine festgelegte Wortbreite des Zählers der Taktteiler berechnet werden. Sei t die Zeitspanne, in der ein Logbuch geführt werden soll, $[0, P]$ mit $P = 2^p - 1$ der Wertebereich des Zählers bei der Bitbreite p und f_T die gesuchte Frequenz für den Zeittakt.

$$\begin{aligned} t &= P\frac{1}{f_T} \\ tf_T &= 2^p - 1 \\ \frac{2^p-1}{t} &= f_T \end{aligned} \tag{54}$$

Beispiel: Soll also $t = 1\,\mathrm{a} = 31536000\,\mathrm{s}$ die zu überwachende Zeit sein und $p = 32$. Damit erhält man eine Frequenz von $f_T \approx 136\,\mathrm{Hz}$. Beim MSP430 existiert der Takt $\mathrm{ACLK} = 2^{15}\,\mathrm{Hz}$ ([7]). Dies ist etwa das 240-fache des oben errechneten Zeittaktes. Sinnvoll erscheint also ein Taktteiler 1:256 bezüglich ACLK. Wie dieses Beispiel zeigt, ist immer noch eine große Wortbreite für das Abspeichern der Zeit nötig. Man kann also nur die Wortbreite des Zählers verringern und damit die Auflösung stark verringern, oder gleichzeitig die zu überwachende Zeit reduzieren. Eine kleinere Zählerwortbreite erscheint aber eine angemessene Lösung zu sein. Verwendet man beispielsweise $p = 16$, so ist $f_T \approx 2 \cdot 10^{-3}\,\mathrm{Hz}$. Damit beträgt die zeitliche Auflösung ungefähr 480 Sekunden.

Da der Zähler bei abgeschaltetem Sensor natürlich nicht arbeitet, sollte der Zählerstand periodisch gesichert werden. Damit entsteht eine zusätzliche, sich aufsummierende Ungenauigkeit nach jedem Einschaltvorgang.

Abseits von der direkten Fehlerbetrachtung kann es sinnvoll sein, auch jeden Systemstart zu protokollieren. Damit kann man folgende Aussagen ableiten:

- Ein häufiges Zusammenbrechen der Versorgungsspannung ist sehr wahrscheinlich eingetreten, wenn der Systemstart wiederholt innerhalb kurzer Zeit aufgetreten ist.
- Eine starke Belastung durch Einschaltvorgänge ist aufgetreten, wenn das System insgesamt sehr häufig neu gestartet wurde.

Ist das Logbuch auf eine Größe angewachsen, dass der zur Verfügung stehende Speicher (bald) nicht mehr ausreicht, sollte in jedem Fall eine Warnung ausgelöst werden. Dies könnte per Interrupt geschehen, sodass eine Software geeignet darauf reagieren kann. Da aber das System oft nicht sofort gewartet werden kann, sind folgende Strategien anwendbar:

- Ist der Speicher endgültig gefüllt und es tritt ein weiterer Fehler auf, kann man den neuen Fehler verwerfen. Damit behält man die Information, ob vielleicht ein früher aufgetretener Fehler der Auslöser für alle Folgefehler sein könnte.
- Will man neu auftretende Fehler nicht verwerfen, so kann man die älteste im Speicher stehende Information überschreiben. Damit kann man die letzten Fehler genau analysieren.

2.9.2.3. Zugriffsmechanismen Möchte man das Logbuch und den Programm-Code für das intelligente Sensorsystem in dem selben nicht-flüchtigen Speicher aufbewahren, muss man dafür Sorge tragen, dass beim Logging nicht der Programmcode gelöscht werden kann. Einerseits sollte die Logbuch-Komponente gar nicht in der Lage sein, auf den Adressbereich des Programm-Codes schreibend zuzugreifen und andererseits existiert die Möglichkeit bei EEPROM, diesen blockweise schreibzuschützen.

Hat man sich für einen seriellen Zugriff auf den nicht-flüchtigen Speicher entschieden, kann dieser somit nur sehr langsam im Vergleich zum normalen RAM-Zugriff im Microcontroller ablaufen. Diese Übertragungsschnittstelle stellt also einen Flaschenhals dar, wenn mehrere Fehler in kurzer Zeit auftreten. Es gilt zu überlegen, wie man diesem Problem begegnet. Man könnte einfach jeden weiteren Fehler ignorieren, oder einen Eingangspuffer aufbauen, der eine gewisse Anzahl von Fehler-Codes aufnimmt. Die andere Alternative wäre, immer eine Rückmeldung zu geben, ob der Fehler angenommen und bearbeitet werden kann - beispielsweise per Interrupt. Möchte man diese Funktionalität auch für Software-Fehlersignalisierung nutzen, so wäre denkbar, in einem RAM-Wort, das im Adressbereich des Microcontrollers liegt und der Logbuch-Komponente zugeordnet ist, ein Signalisierungswort abzulegen, dass anzeigt, ob ein Fehler angenommen werden kann, oder nicht. Der Software-Algorithmus muss dann nur das Datum an dieser Adresse überprüfen und so lange in einer Schleife verharren, bis er Schreibzugriff auf das Fehler-Eingangsregister bekommt.

Weiterhin ist im Einzelfall zu überprüfen, wie das Logbuch (durch Service-Techniker) wieder gelöscht werden kann und ob vielleicht einzelne Logbuch-Einträge modifiziert werden dürfen. Die sicherste Alternative stellt ein Hardwareinterface dar. Dies erhöht aber den pin-count des Prozessors. Daher erscheint eine Softwarelösung ressourcenschonender. Beispielsweise könnten derart Operationen in Software in Verbindung mit einem „Administrator-Passwort“ ausgeführt werden. Dies ist aber sicherheitstechnisch bedenklich.

2.9.3. Interrupt-Auslösung im Fehlerfall einer Komponente

Da einige Möglichkeiten zur Fehlerdetektion in Hardware zu realisieren sind, sollte eine Möglichkeit gefunden werden, geeignet darauf in Software zu reagieren, da ein simples Abschalten im Fehlerfall nicht akzeptabel ist. Durch Softwareroutinen sind weit differenziertere Signalisierungsmöglichkeiten vorhanden. Um dies nutzen zu können, muss dem Prozessor der Fehlerfall signalisiert werden. Am einfachsten ist dies durch die Auslösung eines Interruptes.

In [8] wird detailliert auf die Implementierung der Interrupt-Kette im MSP430 eingegangen. Da mehrere Fehlerquellen eine Signalisierung auslösen können sollen, aber die Interrupts begrenzt sind, liegt es nahe, einen einzigen Interrupt für alle Fehlerfälle zu benutzen. Um dann zwischen den Fehlern unterscheiden zu können, muss die Komponente, die den Interrupt auslöst, nur in den RAM-Bereich des MSP430 eingebunden werden und an einer der Software bekannten Adresse einen Code ablegen, der die Art des Fehlers beschreibt.

Per Software kann der Interrupt maskiert werden. Möchte man auch einzelne mögli-

che Fehlerquellen maskieren, muss man ein Kommunikationsprotokoll zwischen MSP430 und der Fehler-Interrupt auslösenden Komponente implementieren, wie es seitens Texas Instruments bei den Ports realisiert wurde ([7], Kapitel 8).

2.10. Beispielsysteme

Im folgenden werden einige Beispiele für intelligente Sensorsysteme und fehlertolerante Architekturen kurz vorgestellt. Nicht eingegangen wird auf Untersuchungen zur Zuverlässigkeit von ganzen Systemen, wie sie in der Robotik, der Prozessüberwachung oder auch in Kraftfahrzeugen („drive by wire“) vorkommen. Diese sytemübergreifenden Ansätze sind in der Vergangenheit stark untersucht worden und bieten auch heute noch ein weites Feld für die Forschung. Meist wird heute aber so ein System nicht mit intelligenten Sensor-Systemen sondern mit einfachen Sensoren aufgebaut. Fehlerdetektion oder Fehlerkompensation wird meist durch Signalverarbeitung in einer höheren Ebene realisiert. Dabei stehen neben den Informationen von einem Sensor (der eventuell redundant ausgelegt ist) auch zusätzliche Informationen von anderen Sensoren zur Verfügung, sodass die auf diesem Forschungsgebiet gemachten Erkenntnisse selten auf intelligente Sensorsysteme übertragbar sind.

2.10.0.1. Siemens Moore 345 XTC Bei dem Sensorsystem Siemens Moore 345 XTC [27] handelt es sich um ein Drucksensorsystem, welches TÜV-zertifiziert ist. Folgende Fehlererkennungsmechanismen wurden implementiert:

- Detektion aller Kurzschlüsse oder Kontaktabrisse am Sensor.
- Verwendung zweier Sensor-Elemente. Liefern diese unterschiedliche Werte, wird in einen Fehlerzustand übergegangen.
- Verwendung eines „current loop“ (4 bis 20 mA) als Ausgang und Überprüfung des current-loop - Ausganges, ob der erwartete Wert anliegt.

Für die TÜV-Zertifizierung wurden Langzeitanalysen bezüglich der Ausfallwahrscheinlichkeit durchgeführt. Damit konnte ein Modell über die Systemzustände als Markov-Modell erstellt und eine Klassifikation der Ausfallsicherheit nach IEC 61508 durchgeführt werden.

2.10.0.2. 1% genaue Absolutdrucksensorfamilie In [1] wird eine Drucksensorfamilie beschrieben. Folgende Mechanismen zur Fehlererkennung und Fehlerkompensation sind vorhanden:

- Scan-Path - Schnittstelle
- Verwendung eines partitionierten Sensor-Arrays und eines Backup-Sensor-Arrays.
- 2 $\Sigma\Delta$-Modulatoren. Der Signalpfad des einen ist mit dem des anderen vertauschbar.

- Fehlererkennung beim Sensor mittels Stimulation des Sensors und Korrelationsdetektion (Matched-Filter).
- Fehlersignalisierung über analoge Spannung als Ausgang.

2.10.0.3. Temperaturmessung mit einer Brückenschaltung In [5] werden Untersuchungen zur Fehlerdetektion in einem Temperaturmesssystem präsentiert. Dabei wurden folgende Ansätze verfolgt:

- Verschiedene Arte von Versorgungsstromdetektion (I_{ddx}).
- Zwischenspeicherung des letzten Messwertes und anschließender Vergleich mit dem folgenden Messwert. (Ganz ähnlich zu Kapitel 2.6.2.)

2.10.0.4. Current Window Es wird ein System, welches einen zu großen oder zu kleinen Stromfluss detektiert in [28] vorgestellt. Es ist damit nicht primär ein sich selbst überwachendes Sensorsystem, sondern ein System, welches einen externen Sensor observiert.

2.10.0.5. IDR IDR (Integrated Diagnostic Reconfiguration) wird in [29] vorgestellt. Dabei handelt es sich um die Idee, vorhandene oder hinzugefügte Redundanz in einem Sensor auszunutzen, indem man an verschiedenen Punkten der Sensorschaltung Möglichkeiten vorsieht, bekannte Signale einzuspeisen und Signale zu messen. Am Beispiel wird dies demonstriert an mehreren Brückenschaltungen, bei denen jeder Knoten entweder mit der Versorgungsspannung oder dem Eingang des A/D-Wandlers verbunden werden kann. Da die elektrischen Eigenschaften der Elemente der Brückenschaltung bekannt sind, können so Aussagen gemacht werden, ob das System noch innerhalb der normalen Toleranzgrenzen arbeitet.

2.10.0.6. Sensor-Stimulation eines Resonanz-Druck-Sensors In [30] wird ein Sensor vorgestellt, der auf Druck mit einer Veränderung der Resonanzfrequenz zweier micromechanischer Bauteile reagiert. Diese zwei schwingenden Teile sind elektrostatisch gekoppelt und oszillieren jeweils in entgegengesetzter Richtung. Eine Stimulation eines der beiden schwingenden Teile wirkt sich direkt auf das jeweils andere aus, sodass auf diesem Wege eine Fehlerdetektion möglich ist.

2.10.0.7. Eine Watchdog-Realisierung Mit [31] wird unter anderem ein Coprozessor für den AMTEL ATmega103(L) vorgestellt, der die Funktion eines Watchdogs beinhaltet. Innerhalb eines festzulegenden Zeitrahmens muss der Hauptprozessor ein Signal an den Watchdog-Coprozessor senden, um zu signalisieren, dass noch alles in Ordnung ist. Damit ähnelt diese Realisierung vom Grundprinzip her stark dem Watchdog des MSP430 [7].

2.10.0.8. Ein Prozessor für den Einsatz im Weltraum [32] skizziert unter anderem den Einsatz des Prozessors Harris 80RH86 in einem Satelliten des „Space-Based Visible Program“. Alle internen Busse sind 3-fach redundant ausgelegt und Adress- sowie Data-Bus sind mit einer 1-fach-Fehler Korrektur - Fehlerschutzcodierung versehen.

3. Das Modellsystem Drucksensor

Im folgenden wird beispielhaft ein Modellsystem für einen Drucksensor beschrieben. Dabei werden konkrete Annahmen gemacht, aber auch allgemeine Aussagen getroffen. Vor- und Nachteile der ausgewählten Prinzipien werden herausgehoben und Alternativen genannt.

3.1. Die Messschaltung

Es wird als Messschaltung eine klassische Brückenschaltung betrachtet, wie in Abbildung 15 illustriert. Brückenschaltungen gibt es üblicherweise in 3 Variationen - je nach-

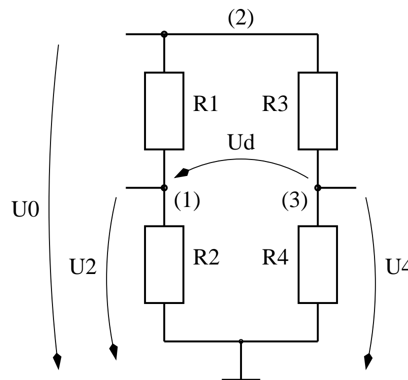


Abbildung 15: Die Brückenschaltung

dem, welche Widerstände sensitive oder passive Elemente sind:

- Ein einziges sensitives Element.
- Zwei sensitive Elemente (R_1 und R_4 oder R_2 und R_3)
- Vier sensitive Elemente, wobei R_1 und R_4 entgegengesetzt sensitiv sein müssen zu R_2 und R_3.

Umso mehr sensitive Elemente eingesetzt werden, desto empfindlicher wird die Brücke bezüglich Änderungen der Messgröße. Die Differenzspannung U_d berechnet sich, wie man leicht nachrechnen kann, zu

$$U_d = U_0 \left(\frac{R_4}{R_3 + R_4} - \frac{R_2}{R_1 + R_2} \right) \tag{55}$$

3.2. Überprüfung der Brückenwiderstände

Eine Einspeisung eines bekannten Stromes an den Knoten 1,2 oder 3 (bei abgeschalteter Versorgungsspannung U_0) bzw. eine Einspeisung der Versorgungsspannung in Knoten 1 oder 3 anstelle in Knoten 2, könnte dazu genutzt werden, die Teilwiderstände der Brückenschaltung zu bestimmen. Diese Einspeisung bringt aber keinen Informationsgewinn auf einem effizienten Weg, denn das Vorgehen hat folgende Nachteile:

- Bei der Methode der Stromeinspeisung muss man mindestens einmal einen Strom in Knoten 1 einspeisen und die Spannungen U_2 und U_4 (bzw. U_d) messen und anschließend das selbe mit einem Strom an Knoten 3 machen, um die Widerstände R_2 und R_4 berechnen zu können. Um auch R_1 und R_3 bestimmen zu können, ist eine Einspeisung an Knoten 2 nötig. Dieses sequentielle Messen bringt einen großen Messfehler mit sich, da sich währenddessen die physikalische Messgröße ändern kann.

- Nach obiger Messung sind zur Berechnung der Widerstände recht komplexe Formeln abzuarbeiten, die mehrere Divisionen beinhalten. Auch muss weiterhin mit erhöhter Genauigkeit gerechnet werden, da die zu erwartenden Ergebnisse für Widerstandswerte von nahe Null bis mehrere MΩ reichen können. Der MSP430 lässt es zwar recht einfach zu, mit 32-Bit-Operanden zu rechnen, aber dies kostet Rechenzeit.

  Prinzipiell ist also eine Berechnung durchaus machbar, aber etwas umständlich.

- Da bei der Bestimmung des Widerstandes eines sensitiven Elementes lediglich auf Plausibilität geprüft werden kann, ist auch keine Aussage über Fehlverhalten möglich, die nicht auch mit Hilfe von (56) und (57) erbracht werden könnte. (Diese zwei Formeln werden in dem Kapitel 3.3.1.1 betrachtet.)

3.2.1. Modifikation der Messbrücke

Speist man Ströme ein bzw. legt man Spannungen zum Testen der Messbrücke an, so ist ein Online-Test sowieso nicht möglich. Besser testbar mittels Offline-Test wird die Messbrücke, wenn man sie dahingehend entwirft, dass jeder der 4 Widerstände „heraus trennbar“ ist. Dies kann mit Hilfe von Schaltern geschehen. Damit und der Möglichkeit, an jedem heraus getrennten Widerstand einen bekannten Strom anzulegen und dann die abfallende Spannung zu messen, ist es möglich, den Widerstandswert einfach und vor allem relativ genau zu bestimmen. An den passiven Elementen ist damit eine recht präzise Aussage über aufgetretene Fehler möglich, während an den aktiven Elementen wieder nur ein Plausibilitätscheck vorgenommen werden kann. Dieses Vorgehen hat eine große Ähnlichkeit mit IDR [29].

Abbildung 16 zeigt eine Möglichkeit einer solchen konfigurierbaren Brückenschaltung. Die Schalter A bis D müssen durch eine geeignete Logik angesteuert werden. Damit ist klar, dass diese Schaltung 4 Versorgungsleitungen für die 4 Knoten und 3 weitere Konfigurationsleitungen benötigt. Die Notwendigkeit für 3 Konfigurationsleitungen ergibt

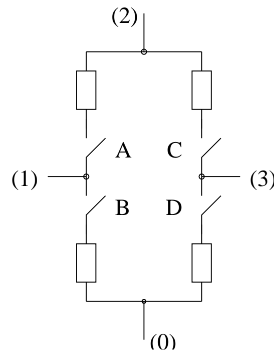


Abbildung 16: Die konfigurierbare Brückenschaltung

sich daraus, dass zwischen normalem Arbeitsmodus und Testmodus umgeschaltet werden muss und nur jeweils einer der Schalter im Testmodus geschlossen ist. Das ergibt 5 mögliche Zustände, die mit 3 Bit codiert werden können. Abbildung 17 illustriert schematisch das I/O-Verhalten einer solchen konfigurierbaren Brückenschaltung.

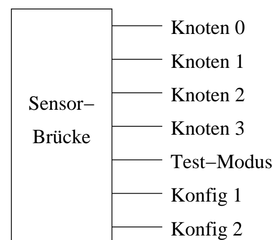


Abbildung 17: Das I/O-Verhalten der konfigurierbaren Brückenschaltung

Mit dieser Testmethode ist keine gute Fehlerabdeckung erreichbar. Detektierbar sind Abrisse aller 4 Signalleitungen, Kurzschlüsse / Abrisse aller Elemente der Brücke und ein Drift der passiven Elemente.

Nicht detektierbar sind schwache Beschädigungen an den sensitiven Elementen, die noch nicht zu einem Verlassen des zulässigen Wertebereichs der einzelnen Messsignale führen.

Diese Methode bietet sich lediglich an, wenn andere Verfahren aufgrund von Messfehlern nur grobe Aussagen zulassen. Auch wäre unter definierten Umgebungsbedingungen eine Selbstkalibration möglich. Problematisch erscheint der mit der Methode einhergehende höhere pin-count.

Um die konfigurierbare Messbrücke ansteuern zu können, ist ein Mux nötig, wie er in Abbildung 18 schematisch dargestellt ist. (Die Konfiguration der Sensor-Brücke wird in diesem Beispiel nicht über den Mux realisiert, sodass die 3 Konfigurationsleitungen entfallen.) Dabei wurde angenommen, dass 2 AD-Wandler, ein Sensor und ein Temperatursensor angeschlossen sind. Auf die konkreten Verbindungen, die dieser Mux schalten muss, soll hier nicht weiter eingegangen werden.

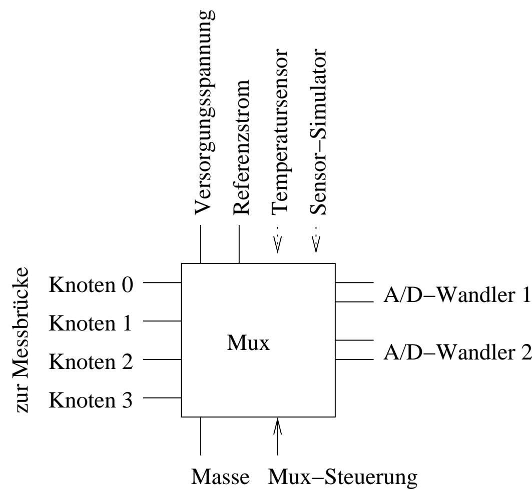


Abbildung 18: Der Mux für die Ansteuerung der Sensor-Brücke

3.2.2. Flexible Ansteuerung der herkömmlichen Messbrücke

Die konfigurierbare Messbrücke bietet den Vorteil, relativ präzise die Teilwiderstände (z. B. zu Kalibrationszwecken) bestimmen zu können, setzt aber voraus, dass direkt am Sensor Veränderungen vorgenommen werden müssen. Will man dagegen bestehende Standard-Sensoren, die auf der einfachen Brückenschaltung basieren einsetzen, so kann auch eine flexible Ansteuerung der Knoten der Messbrücke zu einer Verbesserung der Testbarkeit führen. Damit kann man versuchen, die Messbrücke von außen so zu beschalten, dass sie ähnlich einer konfigurierbaren Messbrücke arbeitet. Es wird daher im folgenden wieder von der einfachen Brückenschaltung (Abbildung 15) ausgegangen.

Die Idee, die durch die konfigurierbare Messbrücke entstanden ist, kann am ehesten dadurch nachgebildet werden, indem in einem Knoten ein bekannter Strom eingespeist oder eine bekannte Spannung angelegt wird und alle anderen Knoten auf Masse liegen. Eine Stromeinspeisung macht aber keinen Sinn, da dann der Strom über 2 Widerstände fließt und somit keine eindeutige Aussage für einen Widerstand möglich ist. Eine Spannungseinspeisung ist also die einzige Möglichkeit. Meist kann dafür die Betriebsspannung verwendet werden.

Legt man nun an Knoten 0, 1 und 3 Masse und an Knoten 2 Betriebsspannung an, so fließen entsprechende Ströme durch die Widerstände R_1 und R_3. Diese kann man mittels I/U-Wandler in eine proportionale Spannung wandeln, welche man mit dem im System vorhandenen A/D-Wandler messen kann. Abbildung 19 illustriert schematisch einen geeigneten I/U-Wandler. Durch den OPV wird die Eingangsleitung auf Masse gezogen. Bei dieser Methode wird es also bei nicht-idealem OPV zu einer geringfügigen Abschwächung der Spannung über dem zu überprüfenden Widerstand der Messbrücke kommen.

Mit dieser Konfiguration können R_1 und R_3 überprüft werden. Möchte man R_2 und R_4 überprüfen, so ist an Knoten 0 Betriebsspannung und die restlichen Knoten Masse

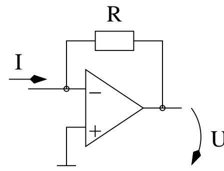


Abbildung 19: Schema eines I/U-Wandlers: $U = -IR$

zu legen.

Da im Endeffekt wieder eine Spannung gemessen wird, muss im Microcontroller der Strom daraus berechnet werden, sodass man mit diesem den Widerstandswert errechnen kann. Bei diesem Prinzip spielt also für die Qualität der Messung die Genauigkeit des I/U-Wandlers eine entscheidende Rolle.

Diese Methode verspricht einen geringeren Aufwand bei der Realisierung, als die in Kapitel 3.2.1 vorgestellte modifizierte Messbrücke und ist gleichzeitig für alle Brückensensoren geeignet. Es wird aber nur die selbe Fehlerabdeckung, wie mit der Realisierung aus Kapitel 3.2.1 erreicht.

3.3. Ausnutzung vorhandener Redundanz

3.3.1. Die herkömmliche Messbrücke

Im folgenden wird von der herkömmlichen Messbrücke, wie sie in Abbildung 15 illustriert ist, ausgegangen.

3.3.1.1. Fehlerdetektion und Lokalisierung Die sensitiven Elemente der Messbrücke kann man näherungsweise modellieren als additive Überlagerung eines festen Widerstandes mit einem Messwiderstand: $R{+}R_{\text{mess}}$. Dabei unterliegt der Wertebereich von R_{mess} gewissen Grenzen, sodass insgesamt ein minimaler R_{min} bzw. ein maximaler Messwiderstand R_{max} erreicht werden kann. Diese Grenzwerte geben automatisch auch die Grenzen für U_d vor. Eine Fehlfunktion ist also sicher eingetreten, wenn U_d nicht innerhalb dieser Grenzen liegt.

Beispiel: Seien R_1 und R_4 die sensitiven Elemente. Sei R_4 in der Form defekt, dass $R_4 < R_{4_{min}}$. Dieser Fehler wird so lange maskiert, wie R_1 groß genug ist, sodass noch eine Spannung U_d gemessen werden kann, die im erlaubten Bereich liegt. Will man diesen Fehlerfall sofort detektieren, muss man zusätzlich U_4 messen. Eine entsprechende Aussage ist möglich, wenn R_1 defekt sein sollte und man dafür U_2 misst.

Die in dem Beispiel angenommene Messung von U_2 und U_4 hat einen weiteren Vorteil: Es ist detektierbar, ob die Spannung von Knoten 1 und 3 innerhalb des jeweils zu erwartenden Wertebereichs liegt. Sollte dies nicht der Fall sein, ist ein Defekt eingetreten, der sich auf beide Seiten der Brückenschaltung ausgewirkt hat. Eine starke Veränderung der Betriebsspannung U_0 wäre so ein Fall. Ein solcher Fehler würde maskiert werden, wenn nur U_d als Messergebnis zur Verfügung steht.

Eine Messung von U_2 und U_4 allein und eine anschließende Differenzbildung, um U_d

zu erhalten, ist messtechnisch aber nicht sinnvoll. Beide Spannungen U_2 und U_4 besitzen nur einen geringen Signalanteil aber einen hohen Offset. Dies würde nach sich ziehen, dass man einen A/D-Wandler benötigt, der eine unnötig hohe Genauigkeit über einen großen Eingangsaussteuerungsbereich besitzt.

Ist die Brücke so abgeglichen, dass bei mittlerem Druck $U_4 = U_2 = U_0/2$ und $U_d = 0\,\mathrm{V}$, so sind alle Widerstände unter diesen Bedingungen gleich. Nimmt man ausgehend von diesem mittleren Druck an, dass die sensitiven Elemente gleichartig in positiver wie negativer Richtung erregbar sind, so gelten folgende Zusammenhänge:

$$U_0 - U_d - 2U_2 = 0 \qquad (56)$$

$$U_0 + U_d - 2U_4 = 0 \qquad (57)$$

Eine Herleitung befindet sich in Anhang D.1.

Liefern diese Prüfgleichungen einen Wert ungleich Null, so ist ein Fehler eingetreten. Somit, ist auch ein Ausfall eines sensitiven Elementes detektierbar, wenn es nicht in der der Form defekt ist, dass R_{min} unter- bzw. R_{max} überschritten wird, also wenn z.B. die Sensitivität (genau) eines Elementes nachgelassen hat. Damit entspricht diese Brückenschaltung in diesem Punkt vom Prinzip her einer 2-Sensor-Anordnung (Kapitel 2.3.2.1). Gleichartige Beschädigungen sind somit wie bei mehr-Sensor-Anordnungen nicht detektierbar.

Es sei darauf hingewiesen, dass das Abgleichen der Widerstände, der Messfehler bei der Bestimmung von U_2 oder U_4 und vor allem die Stabilität der Betriebsspannung U_0 selbstverständlich bei der Überprüfung der Bedingungen (56) und (57) beachtet werden müssen.

Aus Anhang D.1, Gleichung (114) kann weiterhin folgende Gleichung hergeleitet werden:

$$U_2 - \frac{U_0}{2} = \frac{U_0}{2} - U_4 \qquad (58)$$

Diese Gleichung basiert auf den selben Grundsätzen, wie (56) und (57) und liefert damit keine neue Aussage, aber sie bietet einen Ansatzpunkt zur Lokalisierung des Fehlers.

Folgende Fehlerarten werden betrachtet:

- Sensitivitätsverlust (Verkleinerung des Wertebereichs eines sensitiven Brückenelementes)
- stuck-at-Fehler (Festklemmen eines sensitiven Brückenelementes mit vollem Sensitivitätsverlust)
- stuck-at-Fehler in Verbindung mit einer gewissen Restsensitivität

Bei allen genannten Fehlerarten kann davon ausgegangen werden, dass der Widerstandswert des defekten Elementes innerhalb des normalen Arbeitsbereiches liegt und somit durch reine Schwellwertbetrachtung nicht detektiert werden kann.

(58) sagt aus, dass bei intakter Brückenschaltung die Spannung U_2 gleich weit von $U_0/2$ entfernt ist, wie der U_4 von $U_0/2$. Betrachtet man den zeitlichen Verlauf dieser

Abstände, so gilt zwischen zwei zeitliche benachbarten Messwerten

$$\left|\left(U_2(t) - \frac{U_0}{2}\right) - \left(U_2(t+\triangle t) - \frac{U_0}{2}\right)\right| = \left|\left(\frac{U_0}{2} - U_4(t)\right) - \left(\frac{U_0}{2} - U_4(t+\triangle t)\right)\right| \quad (59)$$

Tritt einer der oben genannten Fehler auf, so muss eine der beiden Seiten von (59) kleiner sein, als die andere. Das bedeutet, dass eine Seite der Brücke weniger sensitiv ist, als die andere. Diese weniger sensitive Seite ist defekt.

Um Falschaussagen möglichst zu vermeiden, sollten die Beträge aus (59) über mehrere Messwerte aufsummiert werden, bis die Summen sich ausreichend stark voneinander unterscheiden.

3.3.1.2. Wertung Es erscheint sinnvoll, mit Hilfe von der Messung von U_2 und U_4 eventuell auftretende Fehler schnell zu detektieren und auch mit Hilfe der folgenden Messungen bestimmen zu können, welches sensitive Element defekt ist. In diesem Fall kann dann das System mit dem jeweils anderen noch funktionerenden sensitiven Element in der Halbbrücke mit verminderter Genauigkeit weiter betrieben werden. Für diesen Notbetrieb muss selbstverständlich eine geeignete Kennlinienkorrektur und Messsignalinterpretation vorbereitet werden.

Da ein paralleles Messen aller drei Messspannungen U_d, U_2 und U_4 zu aufwändig erscheint, können die Spannungen mittels sample & hold - Schaltungen gepuffert werden, sodass ein sequentielles Messen möglich wird.

Das große Problem bei dieser Prüfmethode ist eine Schwankung der Betriebsspannung U_0. Sie wird als Fehler durch (56) und (56) detektiert, aber mit (59) wird keine Seite der Brücke als fehlerhaft detektiert. Das System verbleibt also in dem Zustand, einen Fehler zwar detektiert zu haben, aber ihn nicht lokalisieren zu können. Da U_d direkt von U_0 abhängt (siehe (55)), ist eine Detektion eines solchen Fehlers zwar willkommen, aber eine Lokalisierung wäre wünschenswert. Diese ist nur durch eine zusätzliche Messung von U_0 möglich.

3.3.2. Eine dritte Halbbrücke

Die folgenden Überlegungen basieren auf einer Idee von [34].

Um bei der Plausibilitätsprüfung der herkömmlichen Messbrücke (Kapitel 3.3.1) die Abhängigkeit von U_0 zu eliminieren, muss eine differenzielle Messung aller Spannungen erfolgen. Dafür wird eine dritte Halbbrücke hinzu genommen, welche außerhalb des Sensors z. B. innerhalb des Mux realisiert werden kann. Abbildung 20 illustriert diese Idee.

3.3.2.1. Fehlerdetektion und Lokalisierung In Anhang D.2 werden unter den selben Voraussetzungen, wie in Kapitel 3.3.1 folgende Prüfgleichungen hergeleitet:

$$U_d = -2U_l \quad (60)$$

$$U_d = 2U_r \quad (61)$$

$$U_r = -U_l \quad (62)$$

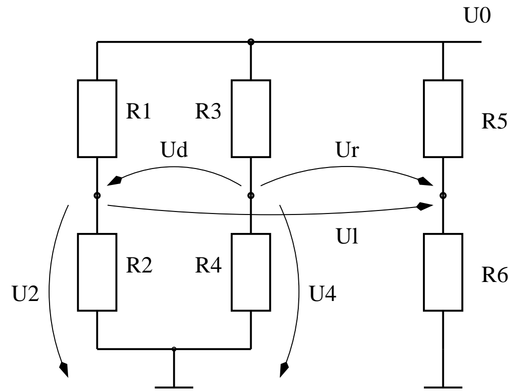


Abbildung 20: Die Brückenschaltung mit einer dritten Halbbrücke

(62) bietet wieder einen Ansatzpunkt, den Fehler zu lokalisieren. Offensichtlich ändert sich U_r um den selben Betrag, wie U_l zwischen zwei zeitlich auf einander folgenden Messwerten. Es gilt somit

$$|U_l(t + \triangle t) - U_l(t)| = |U_r(t + \triangle t) - U_r(t)| \tag{63}$$

Im Fehlerfall ist eine der beiden Seiten von (63) kleiner als die andere. Wieder sollte man, um Falschaussagen zu vermeiden, diese Beträge über mehrere Messwerte aufsummieren, bis die Summen sich ausreichend stark unterscheiden.

3.3.2.2. Detektierbare Fehler Für die Brücke mit zusätzlicher dritter Halbbrücke sollen beispielhaft die Prüfbedingungen für die detektierbaren Fehler angegeben werden.

Fehler	Nr.	U_d	U_r	U_l
Abriss U_0	1	0 V	$< U_{r_{min}}$	$< U_{l_{min}}$
Abriss GND	2	0 V	$> U_{r_{max}}$	$> U_{l_{max}}$
Abriss Pin links	3	0 V*	kein Fehler ($\neq$ 0 V)	0 V*
Abriss Pin rechts	4	0 V*	0 V*	kein Fehler ($\neq$ 0 V)
Kurzschluss R_1	5	$< U_{d_{min}}$	kein Fehler	$> U_{l_{max}}$
Abriss R_1	6	$> U_{d_{max}}$	kein Fehler	$< U_{l_{min}}$
Kurzschluss R_2	6	$> U_{d_{max}}$	kein Fehler	$< U_{l_{min}}$
Abriss R_2	5	$< U_{d_{min}}$	kein Fehler	$> U_{l_{max}}$
Kurzschluss R_3	7	$> U_{d_{max}}$	$> U_{r_{max}}$	kein Fehler
Abriss R_3	8	$< U_{d_{min}}$	$< U_{r_{min}}$	kein Fehler
Kurzschluss R_4	8	$< U_{d_{min}}$	$< U_{r_{min}}$	kein Fehler
Abriss R_4	7	$> U_{d_{max}}$	$> U_{r_{max}}$	kein Fehler

Anmerkungen:

- * : Hochohmiger Eingang am ADC → 0 V.
- Einige Fehlercharakteristika werden durch verschiedene Ursachen ausgelöst (siehe z. B. Fehler 5). Da die verschiedenen Ursachen aber immer innerhalb der selben Halbbrückenseite liegen, kann trotzdem zuverlässig die defekte Seite der Brücke bestimmt werden.
- Einige Fehler sind gleichbedeutend für eine Aussage über die Weiterfunktion der Messbrücke (siehe z. B. Fehler 5 und 6).

Weiterhin sind Sensitivitätsverlust und stuck-at-Fehler (und die Kombination aus beiden) mit Hilfe von (60),(61), (62) und (63) detektierbar und lokalisierbar, sofern es sich nicht um gleichartige Beschädigungen handelt. Ein Sensitivitätsverlust als Defekt auf der linken Brückenseite sei Fehler Nummer 9 und ein solcher Defekt auf der rechten Seite Fehler Nummer 10.

Diese Prüfbedingungen lassen ebenfalls eine Aussage zu, falls die Betriebsspannung U_0 oder die Masseleitung an der Messbrücke nicht das gleiche Potential hat, wie an der dritten Halbbrücke. Dieser Fehler ist allerdings nicht lokalisierbar. Tritt der Effekt stärker auf, so überschreiten U_r und U_l die zulässigen Grenzwerte. Der Extremfall des Kontaktabrisses wurde in der obigen Tabelle schon erfasst.

Mit einem Test zur maximal zulässigen Änderung des Messsignals (Kapitel 2.6.2) ist ebenfalls ein Fehler detektierbar. Wenn man aber nicht gleichzeitig mit Hilfe einer der oben genannten Prüfbedingungen ebenfalls zu einer Aussage kommen kann, ist keine Fehlerlokalisierung möglich.

Eine generelle Variation der Betriebsspannung U_0 ist mit diesen hier genannten Prüfbedingungen nicht detektierbar. Sie wird durch die differentielle Messmethode maskiert.

Die Fehler 3 bis 7 werden nach gewisser Verzögerungszeit auch durch die Prüfung auf Sensitivitätsverlust (Fehler 9 bzw. 10) detektiert, was im Endeffekt die selbe Aussage über den Zustand der Messbrücke zulässt. Umso schneller sich die physikalische Messgröße ändert, desto eher kann Fehler 9 bzw. 10 detektiert werden. Damit sind die Prüfungen auf Fehler 3 bis 7 nur dann nötig, wenn man sie möglichst schnell detektieren will. Die dafür verwendete Rechenzeit ist der einzige Preis für die schnelle Detektion.

3.3.2.3. Wertung Mit Hilfe der dritten Halbbrücke, konnte der Einfluss von U_0 auf die Prüfung der Plausibilität entfernt werden. Damit kann unabhängig von der Betriebsspannung eine Aussage getroffen werden, ob die Brücke korrekt arbeitet.

Die Prüfgleichungen für diese Aussage sind gegenüber der einfachen Brückenschaltung weiter vereinfacht worden und für den Fehlerfall bieten (60) bzw. (61) gleichzeitig sehr einfache Möglichkeiten, mit der jeweils noch intakten Brückenhalbseite weiter zu arbeiten. U_r und U_l können mit dem selben A/D-Wandler gemessen werden wie U_d, sodass ein aufwändiges Umschalten von Offset und Aussteuerbereich des A/D-Wandlers

entfallen kann. Die beiden differentiellen Seitenspannungen weisen dann nur den doppelten Messfehler auf, wie U_d (welcher im Normalfall einen Fehler von $\pm\frac{1}{2}$ LSB hat). Damit kann das System mit hoher Genauigkeit weiter betrieben werden.

Auf der anderen Seite ist durch die Eliminierung des Einflusses von U_0 auf die Prüfgleichungen natürlich eine Möglichkeit entfallen, eine Schwankung von U_0 zu detektieren. Da U_0 multiplikativ in die Berechnung von U_d eingeht (siehe (55)), wirkt sich der Fehler von U_0 additiv auf den Gesamtfehler von U_d aus. (Für U_l und U_r gilt dies analog.) Es ist also sehr wohl nötig, über die Stabilität von U_0 eine Aussage zu machen. Eventuell kann aber eine Schwellwertentscheidung ausreichend sein.

Denkbar ist eine zusätzliche Prüfung von U_0, beispielsweise indem die Spannung über R_6 gemessen wird, die $U_{R_6} = \frac{U_0}{2}$. Dafür ist nun wieder eine Offset- und Aussteuerbereichsumstellung am A/D-Wandler nötig, wenn U_{R_6} mit dem selben A/D-Wandler gemessen werden soll, wie alle anderen Spannungen. Generell steht man vor der Frage, wie der eingesetzte A/D-Wandler auf Schwankungen der Betriebsspannung reagiert. Damit ist eventuell nur eine kleine Schwankungsbreite zulässig. Bei Überschreiten der Grenzbetriebsspannung sollte also das System gestoppt bzw. zurückgesetzt werden. (vergleiche dazu Kapitel 2.2)

Hat man U_0 bestimmt, so kann man daraus direkt den sich ergebenden Fehler für U_d ableiten.

Eine andere Möglichkeit besteht, wenn es möglich ist, die (sinnvoll geteilte) Betriebsspannung als Referenzspannung für den A/D-Wandler zu verwenden und bezüglich dieser Referenzspannung die Quantisierung vorzunehmen. Damit wird der Einfluss der Betriebsspannung auf die Messspannungen eliminiert. Rauschen hat mit zunehmend kleinerer Betriebsspannung einen immer größeren Einfluss auf das Messsignal und damit sinkt natürlich die Qualität der Messwerte.

3.4. Rekonfiguration der herkömmlichen Messbrücke

Wird ein Abriss von Leitungen zum Brückensensor detektiert, so kann ein Abriss an Knoten 1 oder 3 kompensiert werden, indem die jeweils andere Halbbrücke zum Messung weiter verwendet wird. Nicht kompensiert werden kann ein Abriss der Versorgungsspannung oder der Masse-Leitung auf diesem Wege. Um einen derartigen Fehler zu kompensieren, muss ein Mux verwendet werden, der die Brücke „rotieren“ kann. Das heißt, dieser Mux muss die Leitungen zu den Knoten zyklisch vertauschen. Siehe dazu Abbildung 21.

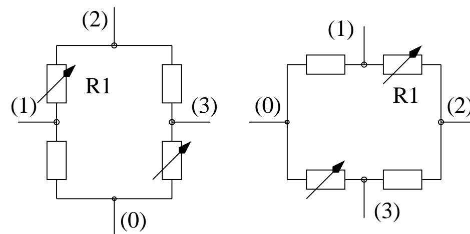


Abbildung 21: Rotation der Messbrücke

Wird also nun die Versorungsspannung an Knoten 1 geleitet anstelle an Knoten 2, Masse an 3 anstelle an 0 und so weiter, so ist der Betrieb einer Halbbrücke weiterhin denkbar, wenn z. B. die Verbindung von Knoten 2 abgerissen ist.

Durch die Rotation ändert sich die Richtung des Anstiegs der zu messenden Spannung. Sollte also im Normalfall ein steigender Druck eine steigende Spannung verursacht haben, wird nun eine fallende Spannung verursacht.

Ob Masse oder die Versorgungsspannung abgerissen ist, kann man nicht immer sagen, wenn man die Brücke in der Normalkonfiguration betreibt - nur dass ein solcher Fehler aufgetreten ist, kann man feststellen. Nach der Rotation ist es aber eindeutig detektierbar, da dann entweder an Knoten 0 oder Knoten 2 eine Spannung anliegt, die nicht innerhalb der Aussteuergrenzen liegt.

Der zusätzliche Hardwareaufwand in Form eines Mux, der die Brücke quasi rotieren kann, lohnt sich umso mehr, je wahrscheinlicher ein Abriss von Leitungen zum Brückensensor ist.

3.5. Das Beispielsystem

Es soll nun ein Überblick über die Architektur eines Beispielsystems gegeben werden. Es wird angenommen, dass entweder die Standard-Messbrücke (Abbildung 15) (eventuell mit Erweiterung durch eine dritte Halbbrücke (Abbildung 20)) oder eine Messbrücke, wie in Abbildung 16 verwendet wird. Schematisch wird das Beispielsystem in Abbildung 22 illustriert. Zu erkennen ist die schon beschriebene Messbrücke und der Mux, sowie ein

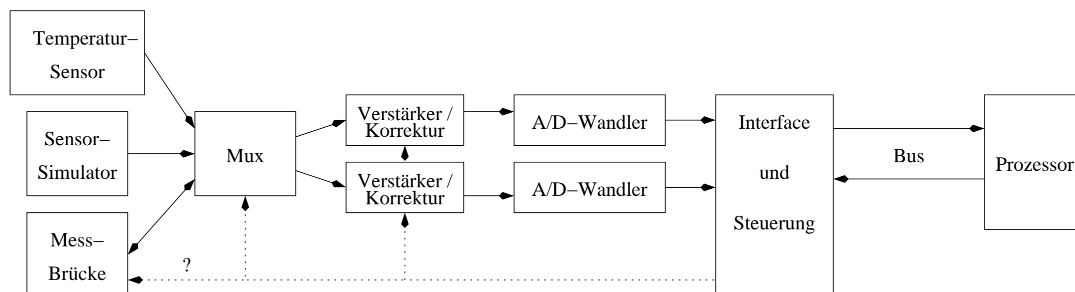


Abbildung 22: Das Beispielsystem in der schematischen Übersicht

möglicher Temperatursensor für eine digitale Temperaturkompensation innerhalb des Prozessors und ein Sensor-Simulator als Referenz für den Test der Mixed-Signal- Signalverarbeitungsstrecke. Es wird angenommen, dass nur eine Brückenschaltung als Sensor angekoppelt ist. Eine sample & hold - Einheit, die es ermöglicht mehrere Teilspannungen der Brücke quasi-parallel zu messen, ist nicht dargestellt. Solch eine S&H-Einheit könnte z. B in den Mux integriert werden.

Weiterhin ist eine zweikanalige Struktur der Signalverarbeitungskette erkennbar. Sowohl der Funktionsblock, der Verstärkerschaltung und analoge Kennlinienkorrektur repräsentiert, als auch der A/D-Wandler sind doppelt ausgelegt. Würde man mehrere Brückensensoren einsetzen, so ist zu überlegen, ob eine Vervielfachung der Signalverar-

beitungsketten Sinn macht, oder ob stattdessen 2 (oder mehr) Verarbeitungsketten im Zeitmultiplex verwendet werden sollten.

Abgeschlossen wird die mixed-Signal-Signalverarbeitungskette durch den Baublock „Interface und Steuerung“, der einen digitalen Schaltkreis darstellt. Er hat folgende Aufgaben:

- Konfiguration der Messbrücke (evtl.), des Mux, der Verstärkerschaltung und der Kennlinienkorrektur, sowie Auswahl der Signalpfade.
- Aufnahme der digitalisierten Daten von den beiden A/D-Wandlern.
- Interface zum Prozessor.

Wird die Brücke mit zusätzlicher dritter Halbbrücke (Kapitel 3.3.2) verwendet, so sollte man im Normalfall Temperatur und Betriebsspannung über den einen Signalpfad und die Brückenspannungen über den anderen messen, damit die Verstärker und A/D-Wandler nicht immer auf einen anderen Aussteuerbereich umgeschaltet werden müssen. Dabei sollte aber die Möglichkeit, dennoch die Signalpfade im Fehlerfall umschalten zu können, nicht entfallen.

Man könnte noch die Möglichkeit einbauen, nach den Verstärkern die A/D-Wandler auswählbar zu machen. Damit und mit dem Sensor-Simulator ist es möglich, festzustellen, ob der Verstärker oder der A/D-Wandler einer Signalverarbeitungskette defekt ist, wenn ein derartiger Fehler aufgetreten ist. Diese Feststellung hilft aber nicht, das System besser weiter betreiben zu können und somit kann diese Umschaltmöglichkeit entfallen.

3.5.1. Das Bussystem zum Prozessor

Für das Bussystem zum Prozessor hin verwendet man ein standardisiertes Übertragungsprotokoll (I^2C, CAN ...), mit dem man einen Kontaktabriss detektieren kann oder man könnte folgendes Interface-Protokoll implementiern:

- Es existieren 2 Datenleitungen, die seriell die digitalisierten Sensordaten zum Prozessor transportieren. Dabei ist es möglich, Daten des A/D-Wandlers 1 über die Leitung 2 zu senden - und umgekehrt. Sollte ein Defekt einer Leitung (Abriss) detektiert werden, werden beide Daten über die noch funktionierende Leitung gesendet. Die Abtastrate würde sich dadurch halbieren. Jeder 2. Messwert würde durch die Interface-Komponente verworfen werden.
- Vom Prozessor wird ein Takt bereitgestellt. Dieser ist so bemessen, dass synchron zu diesem Takt Datenworte über die Datenleitungen gesendet werden können.
- Um einen Kontaktabriss zu detektieren und anzuzeigen, dass ein neues Datenwort gesendet wird, wird vor jedem Datenwort die Bitfolge $\{1, 0, 1\}$ übertragen. Damit muss ein A/D-Wandler, der n Bit Auflösung erreicht, mindestens innerhalb von $n + 3$ Takten ein neues Datenwort liefern.

- Die Steuerungskomponente wird beim globalen Reset (nicht in Abbildung 22 enthalten) mit einer Standard - Einstellung initialisiert. Über eine dritte serielle Leitung kann eine alternative Konfiguration des Sensors übertragen werden. Die Signalfolge $\{1, 0, 1\}$ auf dieser Leitung zeigt an, dass ein neues Wort an Konfigurationsdaten übertragen wird. (Dabei wird angenommen, dass logisch 0 der Zustand ist, wenn keine Konfigurationsdaten gesendet werden.)

Ein derartiges Interface-Protokoll macht es möglich, dass beliebige Arten von A/D-Wandlern einsetzbar sind. Die einzige Forderung ist, dass ein Datenwort innerhalb $n + 3$ Takten bereit stehen muss. Das Datenwort kann in einem Puffer-Register in der Interface-Komponente abgelegt werden. Es bietet sich an, ein Pipeline-Prinzip beim Ablegen zu verwenden: In einem Register wird ein Datum gesammelt und später, wenn es komplett ist, auf ein Ausgangs-Register kopiert. Aus diesem wird gelesen, wenn die Daten seriell über die Datenleitung übertragen werden.

In dem Fall, dass die Übertragungsstrecke mindestens doppelt so schnell arbeitet, wie die A/D-Wandler kann auch nur eine Datenleitung ausreichend sein. Daher verliert man aber die Möglichkeit, beim Abriss der Übertragungsleitung das System weiter betreiben zu können.

Damit sind 6 Verbindungsleitungen vom Prozessor zum Mixed-Signal-Interface nötig: 2× Daten, 1× Konfiguration, der Takt, Betriebsspannung und Masse. Alle Leitungen arbeiten unidirektional.

Bei unidirektionaler Datenübertragung gleich mit welchem Busprotokoll kann ein Abriss detektiert werden, wenn nach einer bestimmten Zeit kein neues Datenwort empfangen wurde. In der Empfangseinheit sollte also ein Timer laufen.

3.6. Detektierbare und nicht detektierbare Fehler

Es wird davon ausgegangen, dass eine Brückenschaltung mit zusätzlicher dritter Halbbrücke im Beispielsystem verwendet wird. (Kapitel 3.3.2). Dabei wird beispielhaft angenommen, dass R_1 und R_4 sensitive Elemente sind.

Direkte Fehler in und an der Messbrücke wurden schon in Kapitel 3.3.2.2 betrachtet. Weiterhin sind folgende Fehler detektierbar:

Fehler	Beschreibung / Detektion
Ausfall eines mixed-signal - Signalpfades	Messwerte überschreiten die Grenzwerte - detektierbar durch Austausch der Signalpfade; Sensor-Simulator; (56) und (57)
Kontaktabriss zum Prozessor	geeignetes Übertragungsprotokoll und Erwarten eines Messwertes nach bestimmter Zeit (Kapitel 3.5.1)

Weitere Überlegungen mit direktem Bezug zur Eigensicherheit des Microcontrollers, sind in Kapitel 2.6 und 2.7 vorgestellt worden.

Nicht von außen detektierbar sind damit Defekte am Temperatursensor, da bis jetzt

nicht angenommen wurde, dass dieser Sensor ebenfalls redundant ausgelegt ist. Ebenso wurde keine Aussage über die Stabilität der Betriebsspannung im mixed-signal - Signalpfad getroffen. Dafür kann nach Kapitel 2.2 eine zusätzliche Komponente zum Test entworfen werden.

Ein Defekt im Mux, der die Sensorsignale in die Signalpfade einspeist ist detektierbar mit Hilfe von Sensor-Stimulation, mit dem Sensor-Simulator oder falls ein Plausibilitätskriterium verletzt wird. (Es könnte beispielsweise das Temperatursensorsingal mit einem Drucksensorsignal vertauscht werden.)

Stuck-at - Fehler sind nicht detektierbar, wenn sie sich gleichartig auf alle Messwerte auswirken, also beispielsweise im Sensor die sensitiven Elemente völlig gleichartig beschädigt werden oder in den Signalverarbeitungsketten die Verstärker oder A/D-Wandler gleichartig von einem Fehler betroffen sind. In diesem Fall kann nur ein weiterer redundanter Sensor mit redundanter Signalverarbeitung, die Sensor-Stimulation oder in begrenztem Maße der Sensor-Simulator den Fehler detektieren. Sollte sich ein Fehler nicht gleichartig auf alle Messwerte auswirken, so ist zumindest eine der Prüfgleichungen aus Kapitel 3.3.2.1 verletzt. Damit ist erst einmal im allgemeinen keine Fehlerlokalisierung möglich, aber auf jeden Fall die Detektion - analog zu einer klassischen 2-Sensor-Anordnung.

Offset-Fehler sind ebenfalls nicht immer detektierbar. Sollten sie sich multiplikativ auswirken (ein Verstärker hat eine zu hohe Verstärkung), so ist dies nur detektierbar, wenn der zulässige Wertebereich für die Messwerte überschritten wird oder sich dieser Fehler nicht auf alle redundanten Messwerte auswirkt. Additive Offset-Fehler sind im allgemeinen schnell durch eine Verletzung der Prüfgleichungen (Kapitel 3.3.2.1) detektierbar, aber oft kann eine Fehlerlokalisierung nicht erreicht werden.

Erratik und Oszillation sind nur detektierbar mit der Sensor-Stimulation oder dem Sensor-Simulator.

Es ist zu erkennen, dass bestimmte Fehlerbilder (z. B. das Überschreiten der Grenzwerte eine Messspannung) mehrere Ursachen haben können (z. B. ein Defekt im Sensor oder in der Signalverarbeitungskette). Es muss also untersucht werden, wo der Fehler aufgetreten ist. Ein derartiger Fehler muss also im Microcontroller ein Programm auslösen, welches zuerst verschiedene Testroutinen startet (z. B. mit dem Sensor-Simulator die Signalverarbeitungskette überprüft).

Hat man einen Fehler detektiert, ist es eventuell möglich, das System (eingeschränkt) weiter zu betreiben.

4. Umsetzung von Lösungsvorschlägen

Im folgenden werden die Realisierungen von ausgewählten Lösungsvorschlägen aus den vorangegangenen Kapiteln vorgestellt. Diese Vorstellung geschieht als Überblick, welcher die Rahmenbedingungen und einige ausgewählte Details der Realisierung beleuchtet.

4.1. Realisierung einer RAM-Fehlerschutzkorrektur

Im folgenden soll eine Umsetzung einer Fehlerschutzkorrektur nach dem in Kapitel 2.7.6.3 aufgestellten (12,8)-Code, der auf einem (15,11)-Hamming-Code basiert, vorgestellt werden. Diese Fehlerschutzcodierung wurde für die VHDL-Beschreibung des Microcontrollers MSP430 [8] in synthesefähiger Form realisiert. Das Modul ist bei der Synthese durch generischen VHDL-Code „abschaltbar“

In Kapitel 2.7.6.7 wurden schon einige grundlegende Überlegungen gemacht, wie der RAM des MSP430 zu organisieren und anzusprechen ist. Im konkreten Fall werden 3 RAM-Blöcke zu je 8 Bit Wortbreite instanziiert. 2 RAM-Blöcke nehmen die normalen Daten auf und der dritte RAM-Block jeweils 4 ECC-Bits der beiden Daten-Blöcke. Illustriert wird dies in Abbildung 23.

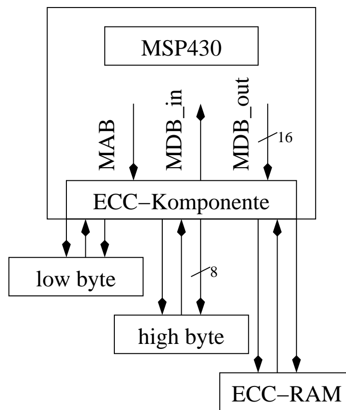


Abbildung 23: Die ECC-Komponente

Für die Simulationen der VHDL-Beschreibung des MSP430 werden RAM-Modelle verwendet. Diese können Software-Programme aufnehmen, indem sie Datenfiles nach dem Intel-Standard (32 bit hex format .hex - siehe z. B. [33]) einlesen. Um für diese Software-Programme die entsprechenden Prüfbits zu generieren, wurde ein nicht synthesefähiges Verhaltensmodell in VHDL entwickelt, welches das jeweilige Datenfile einliest und ein entsprechendes Datenfile für den RAM-Block der Prüfbits generiert. Diese Generierung der Prüfbits geschieht automatisch bei jedem Start des VHDL-Simulators.

4.1.1. Beschreibung der Funktion der ECC-Komponente

Beim Schreiben auf den RAM werden durch die ECC-Komponente die zugehörigen Prüfbits errechnet und an den dritten RAM-Block gesendet. Dazu „lauscht“ die ECC-Komponente auf dem Bus vom Microcontroller zum RAM.

Für das Lesen vom RAM wurde die ECC-Komponente transparent zwischen RAM und Microcontroller eingebettet. Alle Lese-Zugriffe laufen über die Fehlerschutzkorrektur. Es wird automatisch das zugehörige Syndrom errechnet und im Falle eines Fehlers wird dieser korrigiert. Es findet immer eine Fehlerkorrektur und nicht nur eine Erkennung statt. Da die maximale Distanz des (15,11)-Hamming-Codes 2 ist, kann jeder Einzelfehler

so korrigiert werden. Doppelfehler werden durch die automatische Korrektur zu einem fehlerhaften Datenwort decodiert.

Da durch die Korrektur nur die gelesenen Daten korrigiert worden sind, aber im RAM weiterhin der Fehler existiert, so kann der Fall eintreten, dass ein weiterer Fehler in dem Datenwort an der selben Adresse hinzu kommt. Diese ist dann nicht mehr fehlerfrei korrigierbar. Mann kann aber zwei Arten von Fehlern unterscheiden: physikalische Defekte (permanente Fehler) und transiente Fehler. Letztere sind vollständig behebbar, indem man einfach genau das korrigierte Datenwort neu schreibt.

Ein Überschreiben des fehlerhaften Datenwortes kann zufällig geschehen, wenn die Software an die selbe Adresse ein neues Datenwort schreibt. Dieses Vorgehen ist aber nicht befriedigend um sicher gehen zu können, dass ein transienter Fehler ausgelöscht wird. Es muss also eine Möglichkeit gefunden werden, einen RAM-Fehler der Software zu signalisieren, sodass ein Überschreiben des fehlerhaften Datenwortes in Software direkt ausgelöst wird. Dies kann mittels Interrupt geschehen.

In Kapitel 2.9 wurde erwähnt, dass eine Komponente entworfen werden kann, die einen IRQ auslöst. Eine solche Komponente kann als „Sammelstelle“ für in Hardware detektierte Fehler fungieren. (Näheres folgt in Kapitel 4.2.) Die ECC-Komponente muss also im Fehlerfall ein Signal an diese Signalisierungs-Komponente senden.

Es ist daher ein Fehler-Flag in die ECC-Komponente implementiert worden, das folgende Eigenschaften besitzt:

- Das Fehler-Flag wird gesetzt, wenn bei einem Lesezugriff ein Fehler detektiert wird.
- Bei einem Byte-Schreibzugriff muss das jeweils nicht geschriebene Byte gelesen werden, um aus der kompletten Information des RAM-Blocks die Prüfbits zu generieren. Damit handelt es sich um einen Lesezugriff auf das jeweils nicht geschriebene Byte. Sollte durch diesen Lesezugriff ein Fehler detektiert werden, wird das Flag gesetzt.
- Ist das Flag gesetzt, wird die Adresse gespeichert, an der der Fehler aufgetreten ist. Dabei reicht es aus, die Adresse des jeweiligen low-bytes zu speichern (LSB=0), da davon ausgegangen wird, dass zur Fehlerbeseitigung je ein volles 16 Bit Datenwort gelesen und wieder geschrieben wird.
- Ist das Flag einmal gesetzt und treten weitere RAM-Fehler an anderen Adressen auf, so werden die weiteren Fehler ignoriert, da lediglich die erste Adresse gespeichert bleibt.
- Das Flag wird wieder gelöscht, sobald ein word-Schreibzugriff auf die Adresse erfolgt, die gespeichert wurde. (Dies kann auch zufällig geschehen.)

Das Fehler-Flag kann in der Fehlersignalisierungskomponente (Kapitel 4.2) maskiert werden. Ist es nicht maskiert, wird ein IRQ ausgelöst.

4.1.2. Hardwareaufwand der ECC-Komponente

Das Errechnen des Syndroms und die eventuelle Korrektur sowie die Signalisierung eines Fehlers haben einen geringen Hardwareaufwand zur Folge. Nach der Synthese wird vom Synopsys Design Analyzer eine Fläche von rund 85000 μm^2 angegeben. (Zum Vergleich: Die Realisierung des MSP430 allein nach [8] ist in etwa 1 mm^2 groß.

Damit ist der Bedarf an zusätzlichem RAM (150% des normalen Bedarfs) nahezu allein maßgebend für den Mehraufwand bei der Realisierung.

Die maximal erreichbare Taktfrequenz der synthetisierten CPU sinkt durch die ECC-Komponente von rund 8,0 MHz auf rund 7,2 MHz. Bei der Bestimmung dieser Taktfrequenzen wurde mit der Netzliste der synthetisierten CPU simuliert, die alle Gatterverzögerungszeiten, aber keine Pfadverzögerungen, die durch die Verbindungsleitungen zwischen den Gattern entstehen, enthält. Die real erreichbare Taktfrequenz wird etwa halb so groß sein, wie die durch die Gatterverzögerungen bestimmte.

4.2. Fehlersignalisierung im MSP430

Zur Signalisierung eines Fehlers wurde eine Komponente entwickelt, die im Adressbereich des MSP430 als word-module eingebettet ist. Sie ist durch generischen VHDL-Code „abschaltbar“. Der Adressbereich kann bei der Synthese frei gewählt werden (Einschränkungen siehe [7]) und liegt mit der Voreinstellung bei 01F0h - 01FFh.

Für die Auslösung eines Interrupts wird ein Signal generiert, dass in die special function registers (SFR) geleitet wird, wo es ein interrupt flag (IFG) steuert. Es können Interrupts von 0 bis 9 gewählt werden. Voreinstellung ist Interrupt 0. Das jeweilige IFG in den SFR wird ausschließlich durch die Fehlersignalisierungskomponente gesetzt. Es ist nicht schreibbar per Software.

Generell kann per general interrupt enable (GIE - im SR) und per speziellem interrupt enable (IE - in den SFR) der Interrupt maskiert werden. Damit aber eine genauere Steuerung möglich wird, kann innerhalb der Fehlersignalisierungskomponente jede Fehlerquelle einzeln maskiert werden. Aktuell ist dafür Adresse 01FEh vorgesehen. Nach dem Reset (PUC) sind alle Fehlerquellen maskiert.

Bit 0 des Datenwortes bei 01FEh regelt das Interrupt-Verhalten beim RAM-Fehler (Kapitel 4.1). Logisch 0 bedeutet dabei, dass der Interrupt maskiert ist. Das Bit 1 dient als IE für einen Fehler des Parity-Checks innerhalb der CPU (Kapitel 4.4).

Bei Adresse 01F2h liegt die Adresse, an der ein RAM-Fehler aufgetreten ist, sodass eine Interrupt-Routine diese Adresse auslesen und das Datenwort an der betroffenen Adresse lesen und zurück schreiben kann.

Im Falle eines Parity-Fehlers in der CPU obliegt es der Software, darauf zu reagieren (siehe dazu auch Kapitel 2.7.4.1).

Diese Fehlersignalisierungskomponente ist zur Zeit nur als „Rumpf“ implementiert. Lediglich die Signalisierungen eines RAM-Fehlers und eines Parity-Fehlers in der CPU sind enthalten. Für Erweiterungen ist diese Komponente offen. Beispiele dafür sind z. B. ein analoger Iddx-Test, eine Fehlersignalisierung eines Bussystems oder eine Ablaufkontrolle der state machine der CPU. Der Flächenbedarf nach der Synthese beträgt in der

momentanten Form rund 45000 μm^2.

Die interrupt service routine (ISR) für die Fehlersignalisierungskomponente folgt als Programm in C für dem MSP430:

```
interrupt[0x0] void (isr_err_sig)(void) // IRQ 0 => IRQ_addr=FFE0
{
  unsigned int code = *((unsigned int*)err_sig_code);
  unsigned int cnt;

  if ((code & 0x0001) != 0 ) // a error detected by ECC-component
  {
    unsigned int err_addr = *((unsigned int*)err_sig_ramaddr);
    // read address (word) from err_sig_ramaddr
    int err_data = *((unsigned int*)err_addr); // read errogenous data
    *((unsigned int*)err_addr) = err_data; // write back (hardware corrected) data
  }
  if ((code & 0x0002) != 0 ) // a parity error somewhere in the data path
  {
    cnt = *((unsigned int*)err_parity_vio_cnt); // read counter of parity errors
    //(=> reset IRQ source)
    *((unsigned int*)err_parity_vio_cnt) = cnt + 1;

    // a suitable reaction has to be written, if too many parity errors have happened
    if ( cnt > 1 )
      *((unsigned int*)0x0120) = 0x0000;
      // WDT-password-violation - reaction at the moment
  }
}
```

Wird die Komponente erweitert oder deren Adresse oder Interrupt verändert, muss auch die ISR entsprechend daran angepasst werden.

Die momentane Reaktion auf einen Parity-Fehler im Datenpfad (vergleiche dazu Kapitel 4.4) ist ebenfalls nur eine beispielhafte Möglichkeit, die im Einzelfall angepasst werden muss. Zur Zeit wird ein Reset durch eine watchdog password violation ausgelöst. Sinnvoll erscheint in jedem Fall, solch einen Fehler an einen angeschlossenen Host zu signalisieren (sofern der Fehler dies zulässt). Da in dem vorgestellten System noch kein Host angeschlossen ist, wurde dies noch vernachlässigt.

4.3. Systemsimulation

Am Beispiel des Modellsystems mit Brückenschaltung mit zusätzlicher dritter Halbbrücke (Kapitel 3) wurde eine abstrakte Systemsimulation durchgeführt. Damit konnte bestätigt werden, dass die betrachteten Fehler detektiert werden können und gegebenenfalls die Fehlerquelle lokalisierbar ist.

Untersucht wurden all die Fehler in und an der Messbrücke selbst, die in Kapitel 3.3.2 Gegenstand der Untersuchungen waren.

Die Systemsimulation beinhaltet

- die Generierung synthetischer Messsignale (ANSI C),

- die Modellierung der mixed-signal - Signalverarbeitungskette (VHDL, nicht synthesefähig),
- den Entwurf einer Input-Komponente für den MSP430, die die ankommenden Daten bereit stellt und einen Interrupt auslöst (VHDL, synthesefähig - muss später an eine reale Signalverarbeitungskette angepasst werden) und
- die Behandlung der Messwerte inklusive Fehlerdetektion und -lokalisierung (C auf dem MSP430).

4.3.1. Simulation der Messschaltung

Da über den Sensor keine genaueren Spezifikationen vorlagen, ist als Basis die Brückenschaltung mit der zusätzlichen dritten Halbbrücke gewählt worden (Abbildung 20). Folgende Parameter wurden willkürlich gewählt:

- $R_2 = R_3 = 200\,\mathrm{k}\Omega$
- Die sensitiven Elemente sind R_1 und R_4. Es gilt $R_1 = R_4 = 200\,\mathrm{k}\Omega \pm 25\,\mathrm{k}\Omega$
- $R_5 = R_6$
- $U_0 = 5\,\mathrm{V}$
- Auf die sensitiven Elemente wirkt die Messgröße in Form einer sinusförmigen Erregung mit einer Frequenz von 500 Hz , die den gesamten möglichen Aussteuerbereich überstreicht.

Mit Hilfe von einem Programm in ANSI C ist das Verhalten dieser Brückenschaltung simuliert worden. Um für die weitere Simulation verwertbare Daten zu erhalten, wurden die simulierten Messspannungen mit folgenden Parametern „abgetastet“:

- Die Abtastfrequenz beträgt 10 kHz.
- Die Auflösung bei der Abtastung beträgt 12 Bit plus Vorzeichen.
- Der so beschriebene theoretische A/D-Wandler ist eingestellt auf einen Arbeitsbereich von $0\,\mathrm{V} \pm 0{,}35\,\mathrm{V}$.

  Der gewählte Arbeitsbereich ist begründet durch die oben gewählten Parameter der Brückenschaltung und den somit maximal möglichen Messspannungen, wenn kein Fehler aufgetreten ist.
- Die Signale werden zyklisch in der Reihenfolge $U_d \rightarrow U_l \rightarrow U_r$ abgetastet.

Mit Hilfe des Simulationsprogramms werden die abgetasteten Messwerte in einer Textdatei abgelegt. Dabei wird zuerst eine Kennung geschrieben, um welchen Messwert es sich handelt, welches Vorzeichen dieser hat und wie groß der Betrag des Messwertes ist.

Dieses Vorgehen ist motiviert dadurch, dass in einem realen System eine Steuerungskomponente existiert, die regelt, wann welcher Messwert durch die Signalverarbeitungskette geleitet wird und dass somit über den Bus zum Prozessor ebenfalls eine solche Kennung vorhanden ist. Die Eingangs-Komponente im Prozessor kann dementsprechend die hereinkommenden Daten sortieren. Das Abspeichern nach Betrag und Vorzeichen hat den Grund, dass es so einfacher ist, den Textdatenstrom mit VHDL wieder einzulesen.

Bei der Abtastung wurden alle 3 Messwerte parallel aufgenommen, wie es durch sample & hold - Schaltungen im realen System ebenfalls näherungsweise realisiert werden kann.

4.3.2. Simulation des Signalpfades

In VHDL wurde eine Simulation des Signalpfades auf einem hohen Abstraktionsniveau durchgeführt. Zur Simulation wurde ein nicht-synthetisierbares grobes Modell aufgebaut.

In diesem Modell wird die Textdatei, die bei der Simulation der Messschaltung (Kapitel 4.3.1) generiert wurde, eingelesen. Dies entspricht einer Schnittselle zu einem A/D-Wandler. Die in der Textdatei enthaltenen Informationen, zu welcher Messspannung der jeweilige Messwert gehört, simulieren eine Steuerungskomponente für den Signalpfad.

Auf Kennlinienkorrekturen oder andere Anpassungen, wie Verstärkungen wurde in diesem idealen Modell verzichtet.

Die so eingelesenen Daten werden direkt über einen simulierten Bus übertragen. Die Empfangseinheit des Bussystems signalisiert, wenn ein neues Datenwort im Empfangspuffer eingetroffen ist, welches abgeholt werden kann.

Weiterhin wurde eine Komponente in VHDL entworfen, die als Schnittstelle zwischen dem simulierten Bussystem und dem Microcontroller MSP430 dient. Da das Bussystem nur simuliert wurde, ist diese Komponente nur eine Basis für eine spätere Realisierung beispielsweise eines Interface zu einem I^2C-Bus.

Ganz analog zu der Fehlersignalisierungskomponente (Kapitel 4.2) wurde die Schnittstellen - Komponente in den RAM-Bereich des MSP430 bei 01E0h - 01EFh (Voreinstellung) eingebettet und es wurde die Möglichkeit implementiert, einen IRQ auszulösen. Die Komponente wurde mit generischem VHDL-Code beschrieben, der eine Anpassung des Adressbereiches und eine Auswahl des IRQ zulässt. Mögliche IRQs sind IRQ 0 bis 9, sofern andere Komponenten diese Ressourcen nicht belegen. Voreinstellung ist IRQ 1. Die Möglichkeit, einen IRQ auszulösen, wirkt sich selbstverständlich auf die special function register (SFR) aus, sodass diese geringfügig angepasst werden mussten, wie schon in [8] beschrieben.

4.3.3. Fehlerbehandlung in Software auf dem MSP430

Passend zur Schnittstellen-Komponente (Kapitel 4.3.2) wurde auf dem MSP430 in C eine Software geschrieben, die die ankommenden Daten liest und im Bedarfsfall einen Fehler lokalisiert.

Es handelt sich um eine Routine im Interrupt. Wird der entsprechende IRQ ausgelöst, so liest die Software bei 01E0h (in der aktuellen Konfiguration) ein Statuswort, welches

signalisiert, welche Messspannung (U_d, U_l oder U_r) als nächster Messwert bereit liegt oder ob das Bussystem einen Fehler meldet.

Danach wird das entsprechende Datenwort gelesen (U_d bei 01E2h, U_l bei 01E4h und U_r bei 01E6h). Da die Simulation nur dazu dient, mögliche Fehlerfälle zu erkennen, werden die gelesenen Messdaten direkt auf eine willkürlich gewählte RAM-Adresse (01D0h) geschrieben. Das hat den Grund, dass im allgemeinen nach einer Datenverarbeitung im Microcontroller die Daten über einen Bus zu einem Host transportiert werden müssen. Ein solcher Bus wird oft als Komponente in den RAM des Microcontrollers eingebunden und es wird eine Aktion (das Übertragen der Daten) durch das Schreiben eines Datenwortes auf eine bestimmte Adresse ausgelöst.

Die Hauptaufgabe der Software besteht darin, die Fehler nach Kapitel 3.3.2.2 zu detektieren. Ist ein Fehler detektiert, wird an eine weitere willkürlich gewählte RAM-Adresse (01D2h) ein Fehlercode geschrieben, der der Fehlernummer aus Kapitel 3.3.2.2 entspricht.

Das Schreiben der Messwerte und der Fehlercodes an eine bestimmte RAM-Adresse hat für die Simulation weiterhin den Vorteil, dass in der Testbench alle Zugriffe auf diese speziellen Adressen protokolliert und in eine Textdatei geschrieben werden können, sodass bei der Simulation relativ leicht die Funktionsweise des simulierten Systems überprüft werden konnte.

Zusätzlich zu den in Kapitel 3.3.2.2 aufgeführten Fehlern wurde auch noch eine Warnmeldung bei Überlast und eine Signalisierung eines Bus-Fehlers, der durch die simulierte Bus-Komponente detektiert wird, hinzugefügt.

Da die physikalische Messgröße als Sinus-Signal mit bekannter Frequenz für die Simulation modelliert wurde (Kapitel 4.3.2), wurde zur Simulation eines Spike-Fehlers (Kapitel 2.6.2.1) gemäß Anhang B.2 der Grenzwert der Differenz zwischen 2 benachbarten Messwerten bestimmt. Eine Korrektur der detektierten Spikes (z. B. durch Mittelwertbildung) wurde bis jetzt nicht implementiert.

Die Fehler wurden simuliert, indem die simulierte Messschaltung (Kapitel 4.3.1) an das spezifische Fehlerbild angepasst wurde. Damit ist diese Simulation sehr theoretisch. Bei allen Betrachtungen wurde davon ausgegangen, dass die Messbrücke optimal abgeglichen ist und der maximale Messfehler einer gemessenen Spannung bei $\pm\frac{1}{2}$ LSB liegt. Ein reales System muss angepasst und die gewonnenen Erkenntnisse noch einmal überprüft werden. Eventuell muss eine größere Toleranz als $\pm\frac{1}{2}$ LSB einkalkuliert werden

Mit dieser Software konnte gezeigt werden, dass alle betrachteten Fehlerfälle detektiert werden können. Sie bildet damit eine mögliche Basis und bietet Anregungen für ein zu entwickelndes reales System.

4.4. Parity im MSP430

Gemäß den Erläuterungen in Kapitel 2.7.4 wurde in den Datenpfad des MSP430 (Abbildung 11, Seite 44) ein Parity-Check (one bit per data word) abschaltbar mit generischem VHDL-Code implementiert.

4.4.1. Details

Bei der hier vorgestellten Implementierung wurde davon ausgegangen, dass der RAM durch einen speziellen Fehlerschutzcode geschützt ist (siehe dazu Kapitel 4.1), also der RAM bei den folgenden Betrachtungen ignoriert werden kann. Weiterhin wird angenommen, dass auf den Datenleitungen hin zum Adress- und Datenbus des RAM keine zusätzliche Schutzfunktionalität nötig ist, da diese Leitungen lediglich einfache Verbindungen sind. Es wird also kein separater Parity-Check auf diesen Leitungen durchgeführt.

Die hier aufgeführten Details stellen einige durch die spezielle Implementierung aufgetretene Besonderheiten dar. Die allgemeinen Grundlagen wurden in Kapitel 2.7.4 vorgestellt und werden hier nicht weiter betrachtet.

4.4.1.1. Die ALU

Die folgenden Details beziehen sich auf Baugruppen innerhalb der ALU. Siehe dazu auch [8].

Der Adder Um den Hardwareaufwand gering zu halten, wurde eine einfache Parity-Vorhersage, basierend auf den bei der Addition entstehenden carrys realisiert. Auf eine separate carry-Berechnung zur Überprüfung der carrys (siehe Kapitel 2.7.4.14) wurde verzichtet. Damit sind keine Fehler bei der carry-Berechnung detektierbar.

Normalerweise beschreibt man in VHDL eine Addition mit `c<=a+b`, um es dem Synthesetool zu überlassen, welche Art Adder in der Schaltung verwendet wird. Dies richtet sich nach Takt- und Flächenvorgaben und kann durch Syntheseparameter auch manuell gesteuert werden.

Da es in VHDL aber keine Möglichkeit gibt, die Carrys bei einer so beschriebenen Addition zu erhalten, musste manuell ein Adder direkt implementiert werden. Mögliche Adder-Strukturen, die sich gut dafür eignen, sind der einfache carry-ripple-Adder und beispielsweise auch der carry-look-ahead-Adder. Da der Adder aber nicht im kritischen Pfad der Schaltung liegt, wurde ein carry-ripple-Adder für die Realisierung gewählt.

Der VHDL-Code wurde so aufgebaut, dass, wenn kein Parity in der CPU implementiert werden sollte, der Adder wieder in der Form `c<=a+b` beschrieben ist.

Die logischen Funktionen In Kapitel 2.7.4 wurde gezeigt, dass beim Befehl `AND` für die Parity-Vorhersage gleichzeitig die Funktionalität von `BIS` (OR) benötigt wird. Ähnliches ist bei `SXT` nötig. Daher wurden die jeweils benötigten Daten durch Multiplexer über OR- bzw. AND-Gatter mit der Wortbreite von 16 Bit geleitet.

Parity-Generatoren An den folgenden Stellen wurden Parity-Generatoren eingesetzt:

- Parity der Carrys und der zusätzlichen Carrys bei der BCD-Addition innerhalb des Adders (2 Generatoren)
- Parity des Ergebnisses (1 Generator)

- 1 Parity-Generator, der über einen Mux mit verschiedenen Daten gespeist wird. Je nach ausgeführtem Befehl sind dies unter anderem die Eingänge src_in und dst_in, sowie die Ergebnisse des 16 Bit - AND- oder des 16 Bit - OR-Gatters.

4.4.1.2. Der Bus MDB_in Da der RAM durch ECC geschützt ist, muss am Bus MDB_in ein Parity generiert werden.

4.4.1.3. Die Register

Das Instruktions-Register Das Register inst_reg ist mit einem Parity-Check geschützt. Auf dieses Register schreibt nur der Bus MDB_in.

TMP1 und TMP2 Die Register TMP1 und TMP2 werden nur kurzzeitig während der Abarbeitung eines Befehls benötigt. Eine Speicherung von Daten über einen längeren Zeitraum findet nicht statt. Somit könnte man auf einen Parity-Check bei diesen Registern verzichten, wenn man annehmen kann, dass nur ein einzelner Fehler in der CPU in diesen Registern auftritt. Dieser könnte auch innerhalb der ALU detektiert werden, wenn der Fehler durch die Rechnung nicht wieder maskiert wird. Aufgrund dessen wurde der Parity-Check bei TMP1 und TMP2 separat abschaltbar realisiert.

Die Benutzer-Register Die Benutzer-Register sind über 2 Ausgänge R_src und R_dst mit den zugehörigen internen Bussen verbunden. An diesen 2 Ausgängen wurde ein Parity-Check implementiert. Damit ist das Auftreten eines Einzel-Fehlers detektierbar, aber es kann mit der vorhandenen Realisierung nicht ausgesagt werden, in welchem Register der Fehler aufgetreten ist. Diese Information ist allerdings im allgemeinen auch nicht von Interesse.

Des weiteren ist ein zusätzlicher Parity-Generator notwendig, denn wenn auf ein Register mit einem byte-access geschrieben wird, kann nicht das Parity des gesamten Eingangsdatenwortes abgespeichert werden, sondern es muss das Parity des low-bytes berechnet werden. Da von außen auf die Register mit einem byte-access nur über den result_bus geschrieben wird, ist ein zusätzlicher Parity-Check nicht sinnvoll, da schon eine Überprüfung in der ALU implementiert ist. Man geht dabei davon aus, dass kein Fehler direkt auf den Datenleitungen auftritt, bzw. ein Fehler spätestens an den Ausgängen der Register später detektiert werden kann.

Für das SR ist ein dedizierter Parity-Generator nötig, da nach einer Rechnung in der ALU unabhängig von allen anderen Vorgängen in den Registern das SR auf den neuesten Stand gebracht werden muss.

4.4.1.4. Der Incrementer des PC Je nachdem, welchen Stellenwert man einem Fehler innerhalb des PC-Incrementers und des PC (Register 0) einräumt, muss man bei dem PC-Incrementer lediglich einen Parity-Generator für das Ergebnis der Addition implementieren, oder eine Parity-Vorhersage zusätzlich einfügen. Die zusätzliche Vorhersage wurde daher separat abschaltbar mit einem carry-ripple-Adder realisiert.

4.4.1.5. Der Incrementer INC Wie schon beim Incrementer für den PC kann zwischen einem simplen Parity-Generator oder einer Parity-Vorhersage gewählt werden. Die Vorhersage basiert wieder auf einem carry-ripple-Adder.

4.4.2. Fehlersignalisierung und Fehlerbehandlung

In der Fehlersignalisierungskomponente wurden Funktionen für die Signalisierung eines Parity-Fehlers implementiert. Es wird dabei nicht unterschieden, an welcher Stelle des Datenpfades ein Parity-Fehler aufgetreten ist. Die Komponente ist im Detail in Kapitel 4.2 beschrieben. Durch die Funktionalität Parity-Fehler anzuzeigen, kommt ein interrupt enable für Parity-Fehler und eine Speicherzelle bei der Adresse 01F4h hinzu. Die Adresse 01F4h kann benutzt werden kann, um die Anzahl der Parity-Fehler zu speichern. Es obliegt der Software nach einer geeigneten Anzahl von Fehlern angemessen darauf zu reagieren. (Vergleiche dazu Kapitel 2.7.4.1)

Zum aktuellen Zeitpunkt wird der Fehler nur registriert, aber es werden keine Maßnahmen ergriffen, da sich diese Maßnahmen zu sehr nach dem jeweils verwendeten Gesamtsystem richten.

4.4.3. Test und Testergebnisse

Aufgrund der großen Anzahl von möglichen Fehlerquellen und Kombinationen von Datenwörtern kann ein Test der Implementation von Parity im Datenpfad nur ausgewählte und repräsentative Fehler einschließen. Beispielsweise wurde für den Test der Register eines willkürlich ausgewählt und willkürlich ein Bit invertiert.

Durch den Test konnte schnell gezeigt werden, dass Fehler, egal an welcher Stelle sie auftreten, oft katastrophale Auswirkungen auf die Abarbeitung des Programms haben. Am offensichtlichsten ist dies, wenn der Fehler direkt den PC betrifft. In einem solchen Fall kann nur noch von einem „Absturz“ des Prozessors gesprochen werden, wenngleich der Prozessor selten wirklich „einfriert“, sondern meist unsinnige Befehle abarbeitet.

Es kann selten eine Aussage darüber getroffen werden, wie sich ein Fehler an einer bestimmten Stelle im Datenpfad auf das Gesamtsystem auswirkt. Die Programmabarbeitung kann betroffen sein oder lediglich der zu verarbeitende Datenstrom. Da aber auch ein Fehler im Datenstrom durch bedingte Sprünge ebenfalls zu Fehlern in der Programmabarbeitung führen kann, dies aber von den eingesetzten Software-Algorithmen abhängt, kann bei keinem Fehler innerhalb des Datenpfades ausgeschlossen werden, dass dadurch nicht ein „Absturz“ oder eine Fehlrechnung des Prozessors ausgelöst wird.

4.4.3.1. Beobachtungen Temporäre Fehler, die sich nur kurzzeitig auf den Datenpfad auswirken, also z. B. ein Bit in den Registern umkippen lassen, verhindern eine Fehlerbehandlung nicht. Sowohl ein Fehlerzähler als auch eine Meldung an einen angeschlossenen Host sind uneingeschränkt möglich.

Permanente Fehler (beispielsweise im PC) können dazu führen, dass nicht einmal mehr die interrupt service routine ISR zur Fehlerbehandlung mehr ausgeführt werden

kann. Die ISR sollte daher so einfach wie möglich aufgebaut sein und auf dem einfachsten möglichen Weg einen Fehler in der CPU an einen Host melden, damit die Abarbeitung der ISR möglichst wenige Teile des Datenpfades benötigt. Um unabhängig von der Funktionalität des Datenpfades zu sein, ist es denkbar in Hardware eine Lösung zu implementieren, mit der auch bei permanenten Fehlern im Datenpfad eine Meldung an den Host abgesetzt wird. Dazu ist die Kommunikationskomponente zum Host geeignet zu modifizieren. Eine solche Hardware-Lösung ist bis jetzt nicht realisiert und kann Teil von weiterführenden Überlegungen sein.

4.4.3.2. Besonderheiten Nicht durch Parity geschützt werden die 4 Flags der CPU, welche in der ALU bei der Abarbeitung eines Befehls generiert und im SR gespeichert werden. Die Flags werden aus dem Ergebnis der ALU-Rechnung durch kombinatorische Logik bestimmt und direkt an das SR zur Speicherung weiter geleitet.

Der PC wird hauptsächlich durch den eigens dafür vorhandenen PC-Incrementer verändert. Sollte ein Fehler in den Latches (die den PC speichern) oder in dem PC-Incrementer auftreten, so kann sich dieser Fehler schnell fortpflanzen und zu Mehrfachfehlern führen, die nicht mehr detektiert werden können. Es erscheint damit sehr sinnvoll, für den PC-Incrementer die Parity-Vorhersage zu verwenden.

Anders als der PC-Incrementer, wird der Incrementer INC im allgemeinen seltener verwendet, sodass ein Fehler sich nicht so schnell fortpflanzt. Weiterhin schreibt INC nur auf die Register, die wiederum durch einen Parity-Check gesichert sind, sodass die gespeicherten Daten geprüft werden, bevor sie für eine folgende Rechnung verwendet werden. Eine Ausnahme bildet der PC (der auch mit dem Ergebnis von INC überschrieben werden könnte), der direkt auf den MAB schreiben kann. Für diese Verbindung ist zum aktuellen Zeitpunkt kein zusätzlicher Parity-Check vorgesehen. Dies alles lässt die Aussage zu, dass eine Vorhersage im Incrementer INC nicht so nötig ist, wie beim PC-Incrementer.

In der ALU bei den Befehlen `CMP` und `BIT` wird der Wert des Destination-Registers nicht verändert. (Der Wert von destination wird als result aus der ALU nach außen geleitet und destination wird mit dem ursprünglichen Wert überschrieben. [8]) Das Ergebnis dieser Befehle sind nur die gesetzten Flags. Somit sind diese beiden Befehle nicht direkt mit einem Parity-Check testbar. (Es wird lediglich geprüft, ob der Wert von destination korrekt durch die ALU „getunnelt“ wird, aber dies hat nichts mit der Funktion der beiden Befehle an sich, sondern nur mit der gewählten Implementierung zu tun.)

4.4.4. Hardwareaufwand

Die folgenden Angaben zur Fläche, die für die Schaltung benötigt wird, gibt das Synthesetool Synopsys Design Analyzer nach dem Synthesevorgang an. Die Werte wurden gerundet. Da die Entwicklung der synthesefähigen Beschreibung des MSP430 noch nicht vollständig abgeschlossen ist, kann die tatsächlich benötigte Fläche sich später ändern.

Die erreichbare Taktfrequenz wurde bestimmt durch Simulation der synthetisierten CPU (Netzliste). Dadurch sind alle Gatter - Verzögerungszeiten, aber keine Pfad - Verzögerungen, die durch die Verbindungsleitungen und deren Kapazität entstehen,

enthalten. Die real erreichbare Taktfrequenz sollte also bei etwa der Hälfte der durch die reinen Gatterverzögungen bestimmten Taktfrequenz liegen.

Auswahl	Fläche/μm^2	max. Takt/MHz
CPU ohne Parity	934000	8,0
CPU mit Parity	1087000	7,5
CPU mit Parity, dabei Parity-Vorhersage in beiden Incrementern	1117000	7, 8

Auffällig ist die höhere erreichbare Taktfrequenz bei der CPU mit Parity-Vorhersage in beiden Incrementern. Da die Incrementer nicht im kritischen Pfad liegen, kann die einzige Erklärung dafür sein, dass aufgrund der expliziten Modellierung der Adder als carry-ribble-adder und damit des Aufbrechens des Adders in seine Teilkomponenten es dem Synthesetool möglich war, das Design besser zu optimieren.

Zu der benötigten Fläche kommt doch die Fläche für die Fehlersignalisierungskomponente hinzu, die aber ausserhalb der CPU in der Beschreibung des Microcontrollers instanziiert ist.

Es erscheint sinnvoll, nicht nur die CPU selber, sondern auch an die CPU angeschlossene Komponenten (RAM-mapped components) mittels Parity zu sichern. Beispiele dafür sind der Multiplizierer, sowie I/O-Komponenten. Bei einer konkreten Realisierung ist dann diese Aufgabe zu lösen. Die dafür benötigte Fläche addiert sich zu der oben angegebenen hinzu.

Setzt man den ECC-RAM (siehe dazu Kapitel 4.1) und Parity im Datenpfad der CPU gemeinsam ein, so kann eine Taktfrequenz von 6,8 MHz erreicht werden.

5. Zusammenfassung und Ausblick

In dieser Arbeit wurden Prinzipien für ein eigensicheres intelligentes Sensorsystem vorgestellt. Das Ziel war die Betrachtung und Entwicklung von Prinzipien, die es ermöglichen einen Fehler im System zu detektieren, darauf zu reagieren und im Idealfall eine Weiterfunktion (bei verminderter Güte der Daten) zu ermöglichen. Das Sensorsystem sollte eine erhöhte Eigensicherheit durch diese Ideen erhalten.

Betrachtet wurden mögliche Fehler am Sensor und speziell an den sensitiven Elementen, bei der Verbindung zu der mixed-signal Signalverarbeitungskette (Verstärker, A/D-Wandler) und Fehler in diesen Komponenten selbst, in dem digitalen Bussystem zu einem Prozessor und in dem Prozessor selbst.

Fehler innerhalb des Sensors und der mixed-signal Signalverarbeitungskette wurden oft allgemein gültig und unabhängig von einem konkreten Beispiel betrachtet. Als Prozessor ist dagegen der Microcontroller MSP430 in der synthesefähigen Beschreibung [8] beispielhaft gewählt worden.

Eine Schwerpunkt bei der Entwicklung von Testkriterien war die Möglichkeit der Fehlerdetektion und Fehlerbehandlung auf dem Microcontroller MSP430. Hardware, die

die Detektion oder die Fehlerbehandlung erleichtert, wurde ebenfalls vorgestellt. Primär stellt diese Hardware aber nur eine Unterstützung des Microcontrollers dar.

Neben den theoretischen Betrachtungen wurden folgende Dinge realisiert:

- eine Fehlerschutzkorrektur für den RAM des Microcontrollers MSP430 nach einem (12,8) Blockcode, der auf einem (15,11)-Hamming-Code basiert und alle Einzelfehler korrigieren kann
- eine Fehlersignalisierungskomponente, die dem MSP430 per Interrupt meldet, dass ein Fehler aufgetreten ist und eine geeignete interrupt service routine, die eine passende Reaktion auslöst
- eine Systemsimulation, die einen Sensor, der auf einer Brückenschaltung basiert, einen Signalpfad, eine Input-Komponente für den MSP430 und Softwareroutinen für den MSP430, welche mögliche Fehler detektieren und lokalisieren umfasst
- eine Absicherung durch Parity auf dem gesamten Datenpfad des MSP430, sodass Einzelfehler detektiert werden können

Alle theoretischen Betrachtungen und praktischen Realisierungen stellen einen Ansatz dar und sollen Anregungen für ein zu entwickelndes reales System darstellen.

Die Realisierung des ECC-RAM für den MSP430 und die Parity-Absicherung des Datenpfades stellen jeweils ein abgeschlossenes Teilprojekt dar, dass ohne weitere Modifikationen eingesetzt werden kann. Lediglich in dem Fall, dass ein anderer Prozessor als der MSP430 für ein eigensicheres Sensorsystem ausgewählt wird, sind sie nur Beispiele und Anregung.

Anders verhält es sich dagegen mit den Softwareroutinen auf dem MSP430, die verschiedene Fehlerfälle in dem Beispielsystem detektieren sollen. Sie zeigen lediglich die Umsetzbarkeit der angewendeten Prinzipien. Sie müssen auf den Sensor und den Signalpfad des zu entwickelnden realen Systems angepasst werden und zusätzlich werden Routinen benötigt, die den Fehlerfall an einen angeschlossenen Host melden, sowie Sensor und Signalpfad geeignet rekonfigurieren, um mit milder Degradation die Messwertaufnahme weiterhin zu ermöglichen.

Die Fehlersignalisierungskomponente ist als voll funktionsfähiger Rumpf vorhanden, die in einem realen System weiter ausgebaut werden kann.

A. REiSO

REiSO = Recalculation with invers shifted Operands. (In Anlehnung an [12].)

Beim Microcontroller MSP430 kann REiSO mit dem Befehl `RRA Rn` und RESO durch `ADD Rn,Rn` als Addition des Operanden mit sich selbst realisiert werden.

Offensichtlich kann bei sämtlichen logischen Funktionen REiSO problemlos angewendet werden, da keinerlei Wechselwirkungen zwischen den Bits in Form von Carrys o. ä. auftreten. Dabei ist ebenfalls genau 1 Bit nicht testbar: das LSB des Ergebnisses. Damit ist REiSO in diesem Bereich zumindest gleichwertig zu RESO. Da häufig die Operanden nicht die volle Wortbreite von 16 Bit benötigen und bei REiSO das LSB, was stets von Bedeutung ist, beachtet wird, kann REiSO geringfügig besser geeignet sein - je nach auftretenden Operanden.

Problematischer wird es bei den arithmetischen Operationen. Es sollen die dortigen Zusammenhänge hergeleitet werden. Dafür wird die folgende Definition benötigt: Es sei $shr(a)$ ein Shift um 1 Bit nach rechts, wie er durch `RRA Rn` auf dem Microcontroller MSP430 realisiert wird. Die Abbildung ist nicht bijektiv.

Jede Zahl $a \in \mathbf{Z}$, binär repräsentiert im Zweierkomplement, kann man somit eindeutig darstellen als:

$$a = 2\,shr(a) + (a \bmod 2) \tag{64}$$

Dabei ist $(a \bmod 2)$ ein Ausdruck, der gleich 1 ist, wenn a ungerade ist und gleich 0 sonst.

Eine nützliche Überlegung ist folgende: Seien $g, h \in \mathbf{Z}$ ganze Zahlen, so gilt:

$$shr(2g + h) = g + shr(h) \tag{65}$$

Dadurch, dass $2g$ eine gerade Zahl ist, ist bei ihrer binären Repräsentation das LSB gleich Null. Somit kann ohne Informationsverlust $shr(2g)$ ausgeführt werden. Die Addition aus (65) kann „ausgeklammert“ werden, da eine gerade Zahl als Summand keine Veränderung des LSB des Ergebnisses auslöst. Die Zahl $shr(h)$ kann möglicherweise nicht im Zweierkomplement dargestellt werden, da h ungerade sein könnte. Für die mathematische Gleichheit der beiden Seiten der Formel muss also dieser Ausdruck unangetastet bleiben. Ziel muss es sein, solcherart Ausdrücke zu eliminieren.

Alle bis hierhin gemachten Überlegungen gelten, wie schon erwähnt auch für binäre Darstellung im Zweierkomplement. Selbstverständlich führt eine Shift-Operation nach rechts angewendet auf eine negative Zahl im Zweierkomplement zu einer positiven:

$$shr(-k) = 2^{n-1} - ((k+1)\,\mathrm{DIV}\,2) \qquad ; -k \in \mathbf{Z},\ -k < 0 \tag{66}$$

Dabei bezeichnet n wieder die zur Verfügung stehenden Bits und k muss selbstverständlich innerhalb des Wertebereichs liegen (12).

A.1. REiSO bei der Addition

Seien $a, b \in \mathbf{Z}$ ganze Zahlen in binärer Repräsentation im Zweierkomplement. Dann gilt mit (64):

$$a + b = (2\,shr(a) + (a \bmod 2)) + (2\,shr(b) + (b \bmod 2)) \tag{67}$$

Damit kann man selbstverständlich noch nicht viel anfangen. Daher soll einmal nach rechts geshiftet werden. Diese Operation ist irreversibel.

$$shr(a+b) = shr[2\,shr(a) + 2\,shr(b) + (a \bmod 2) + (b \bmod 2)] \tag{68}$$

Mit Hilfe von (65) kann man dies vereinfachen.

$$shr(a+b) = shr(a) + shr(b) + shr[\underbrace{(a \bmod 2)}_{A} + \underbrace{(b \bmod 2)}_{B}] \tag{69}$$

Zur Erinnerung: Die Ausdrücke A und B sind entweder 1 oder 0.

A+B	0	1
0	0	1
1	1	2

Nur in dem Fall, wo das Ergebnis der Addition 2 (also binär: 10) ist, hat dies eine Auswirkung auf das Ergebnis nach dem Shift nach rechts. Somit kann man (69) vereinfachen und erhält:

$$shr(a+b) = shr(a) + shr(b) + A \wedge B \tag{70}$$

Bei dieser Darstellung wurde bewusst eine Mischung aus Arithmetik und Logik verwendet, da somit ersichtlich ist, dass hier ein `ADDC` der geeignete Befehl für die Ausführung ist. Das Carry-Flag berechnet sich zu $C = A \wedge B$, wobei noch einmal erwähnt werden soll, dass A und B jeweils das LSB von a und b darstellen.

A.2. REiSO bei der Addition mit Carry

Der Befehl `ADDC` kann analog wie `ADD` behandelt werden. Es ist leicht zu sehen, dass mit (69) folgt:

$$shr(a+b) = shr(a) + shr(b) + shr[\underbrace{(a \bmod 2)}_{A} + \underbrace{(b \bmod 2)}_{B} + C] \tag{71}$$

A+B+C		0	1
0	0	0	1
0	1	1	2
1	1	2	3
1	0	1	2

Ob das Ergebnis der Summation 2 oder 3 ist, spielt nach dem Shift nach rechts keine Rolle mehr. Das Endergebnis ist also:

$$shr(a+b) = shr(a) + shr(b) + [(A \vee B)C \vee AB] \tag{72}$$

Dabei wurde für die bessere Übersichtlichkeit das AND=$\wedge$ wie der Punkt bei einer Multiplikation weggelassen, wie es häufig in der Literatur geschieht.

Auch hier kann wieder die Operation `ADDC` verwendet werden. Die Berechnung des Carry-Flags ist allerdings etwas aufwändiger bezüglich der Rechenzeit.

A.3. REiSO bei der Subtraktion

Es gelten die selben Definitionen, wie auch bei der Addition. Der Rechenweg ist analog.

$$a - b = (2\,shr(a) + (a \bmod 2)) - (2\,shr(b) + (b \bmod 2)) \tag{73}$$

$$shr(a - b) = shr(a) - shr(b) + shr(\underbrace{(a \bmod 2)}_{A} - \underbrace{(b \bmod 2)}_{B}) \tag{74}$$

A-B	0	1
0	0	-1
1	1	0

In dem Fall, wo das Ergebnis der Subtraktion -1 ist, muss $2^{n-1} - 1$ zum Ergebnis dazu addiert werden - siehe (66). Damit ergibt sich das endgültige Ergebnis zu

$$shr(a - b) = shr(a) - shr(b) + (\overline{A}B)(2^{n-1} - 1) \tag{75}$$

Es sind also mehrere Operationen nötig: `SUB` und `ADD`, sowie natürlich die Prüfung, ob die Addition ausgeführt werden muss.

A.4. REiSO bei der Subtraktion mit Carry

Der Befehl `SUBC a,b` $\equiv a + \overline{b} + C$ bildet die mathematische Operation $a - b - \overline{C}$ ab.

$$a - b - \overline{C} = (2\,shr(a) + (a \bmod 2)) - (2\,shr(b) + (b \bmod 2)) - \overline{C} \tag{76}$$

$$shr(a - b - \overline{C}) = shr(a) - shr(b) + shr(\underbrace{(a \bmod 2)}_{A} - \underbrace{(b \bmod 2)}_{B} - \overline{C}) \tag{77}$$

$A - B - \overline{C}$		0	1
0	0	-1	0
0	1	-2	-1
1	1	-1	0
1	0	0	1

Nur die negativen Ergebnisse wirken sich nach dem Shift nach rechts aus. Damit ergibt sich mit (66) und $shr(-2) = shr(-1)$:

$$shr(a - b - \overline{C}) = shr(a) - shr(b) + (A\overline{B} \vee (\overline{A} \vee B)\overline{C})(2^{n-1} - 1) \tag{78}$$

Es sind also genau wie bei der normalen Subtraktion ein `SUB` und eventuell ein `ADD` nötig zur Berechnung.

A.5. REiSO bei der Multiplikation

Es gelten die selben Definitionen, wie bei der Addition und Subtraktion. Verwendet man den analogen Rechenweg, so erhält man:

$$\begin{aligned} ab &= (2\,shr(a) + (a \bmod 2)) * (2\,shr(b) + (b \bmod 2)) \\ &= 4\,shr(a)shr(b) + (a \bmod 2)(2\,shr(b)) + (b \bmod 2)(2\,shr(a)) + \\ &\quad +(a \bmod 2)(b \bmod 2) \end{aligned} \tag{79}$$

Der Gedanke liegt nahe, ein Shift nach rechts auszuführen. Dabei wird der letzte Teil der Summe wegfallen, da durch die Multiplikation dort maximal 1 entstehen kann.

$$shr(ab) = 2\,shr(a)shr(b) + (a \bmod 2)shr(b) + (b \bmod 2)shr(a) + 0 \tag{80}$$

Das Ergebnis ist nicht wirklich befriedigend, da eine Kontrollrechnung nach diesem Schema relativ aufwändig wäre. Sie würde eine Multiplikation mit anschließendem Shift um eine Stelle nach links und eventuell zwei weitere Additionen mit den nach rechts geshifteten Operanden.

Versucht man das Problem zu lösen, indem man noch einmal nach rechts shiftet, erhält man folgenden Ausdruck:

$$shr(shr(ab)) = shr(a)shr(b) + shr[(a \bmod 2)shr(b) + (b \bmod 2)shr(a)] \tag{81}$$

Der letzte Summand lässt sich nicht weiter vereinfachen und man hat damit keine Vereinfachung gegenüber (80) erreicht.

A.6. RESO im Vergleich REiSO

Bei rein logischen Operationen ohne Wechselwirkungen zwischen den einzelnen Bit-Spalten nehmen sich beide Verfahren nicht viel. Lediglich, wenn hauptsächlich mit Datenworten gerechnet wird, die betragsmäßig eine relativ kleine Zahl darstellen (wo das MSB also immer Null ist), erreicht RESO die bessere Fehlerstimulation, da das LSB überprüft werden kann.

Bei allen anderen Operationen ist RESO eindeutig besser geeignet als REiSO, da die Kontrollrechnung wesentlich einfacher gehalten werden kann. Das LSB des ursprünglichen Operanden bleibt immer bestehen und fließt korrekt in die Kontrollrechnung ein.

REiSO leidet an der aufwändigen Berechnung der Carrys (Addition) und den eventuell zusätzlichen Summationen (bei den Subtraktionen und der Multiplikation).

Beispielsweise könnte folgender Assembler-Code für die Addition und REiSO verwendet werden, wenn `Ra` und `Rb` die Register mit Kopien der Inhalte der Operanden a und b sind:

```
MOV  Ra,Ra_c
MOV  Rb,Rb_c
RRA  Ra          ; shr(a)
RRA  Rb          ; shr(b)
```

```
AND  Ra_c,Rb_c
RRA  Rb_c        ; set carry-flag
ADDC Ra,Rb
```

In `Rb` liegt dann der Wert, der mit $shr(a + b)$ verglichen werden muss. Damit sind im Gegensatz zu RESO 2 Kopieroperationen, die AND-Verknüpfung und eine zusätzliche Rotation nötig.

Noch ungünstiger wird das Verhältnis bei der Addition mit Carry. Hier wird etwas mehr Logik benötigt um das Carry-Flag zu berechnen. Bei den Subtraktionen ist das Verhältnis ähnlich, wie bei der Addition mit Carry, da eine zusätzliche Addition eventuell benötigt wird. Bei der Subtraktion mit Carry tritt eine weitere Verschlechterung ein, da die Logik für die Entscheidung etwas aufwändiger geworden ist.

Bei der Multiplikation ist das Verhältnis zwar nicht schlechter, als bei der Subtraktion mit Carry, aber damit ist ebenfalls RESO deutlich effizienter.

Weiterhin ist natürlich die Voraussetzung für eine schnelle Abarbeitung von REiSO, dass genügend Register frei sind, um mit wenigen Operationen die Kontrollrechnung durchzuführen. Selbiges gilt zwar auch für RESO, aber es werden weniger Register benötigt.

Abschließend kann man sagen, dass REiSO generell meist deutlich schlechter als RESO zu implementieren ist. Der kleine Vorteil von REiSO, nur bei der Multiplikation Probleme mit den Grenzen des Zahlenbereichs zu bekommen, wiegt im allgemeinen nicht die höhere Rechenzeit auf.

B. Abschätzung der maximalen Signaländerung

Im folgenden soll eine Abschätzung vorgenommen werden, wie stark sich zwei benachbarte Messwerte eines Signals bei bekannter Abtastfrequenz unterscheiden dürfen. Diese Grenze ist ein Kriterium für transiente Fehler.

Die folgenden Rechnungen sind einheitenlos. Es soll das Prinzip für den allgemeinen Fall dargestellt werden.

B.1. Ein bekannter typischer Signalverlauf

Ist ein typischer Signalverlauf bekannt, so ist die Berechnung der maximal möglichen Änderung zwischen 2 Messwerten einfach. Sei $x(t)$ das kontinuierliche und $x(k\triangle t)$ das mit $f_A = 1/\triangle t$ abgetastete Messsignal. Dabei wird mit $k \in \mathbf{N}$ der k-te Abtastwert bezeichnet.

Die maximale Signaländerung tritt auf, wenn der Anstieg $x'(t)$ maximal wird. Das Kriterium dafür ist also $x''(t) = 0$, was dem Wendepunkt der Kurve entspricht.

Das abgetastete Signal $x(k\triangle t)$ unterliegt der stärksten Änderung, wenn 2 Abtastpunkte $k_1\triangle t$ und $k_2\triangle t$ um den Wendepunkt herum gleich weit entfernt liegen. Sei t_W der Wendepunkt, so tritt der worst-case ein, wenn der erste Abtastwert bei $t_W - \frac{1}{2}\triangle t$ und der zweite bei $t_W + \frac{1}{2}\triangle t$ liegt.

Damit berechnet sich die maximale mögliche Signaländerung nach:

$$\triangle x(k\triangle t) = \left| x(t_W - \frac{1}{2}\triangle t) - x(t_W - \frac{1}{2}\triangle t) \right| \tag{82}$$

Das Problem ist es, einen bekannten typischen Signalverlauf zu finden. Im allgemeinen kann also mit diesem einfachen Ansatz keine Aussage gemacht werden. Die gewonnenen Erkenntnisse werden aber in den folgenden Abschnitte genutzt.

B.2. Ein Signal mit einer charakteristischen Frequenz

Errechnet man die Fourier-Reihe (FR) des analogen Signals, das die Messgröße repräsentiert, bzw. errechnet man die Diskrete Fourier Transformation (DFT) des entsprechenden abgetasteten Signals, so erhält man eine Spektrum (das periodisch ist bei der DFT) welches eine maximale Frequenz enthält (bei der DFT: eine maximale Frequenz im Basisband). Sei f_0 näherungsweise die alleinig bestimmende Frequenz dieses Signals und sei A dessen Amplitude.

Das Frequenz-Spektrum des Signals kann also also mit dieser Näherung wie in Abbildung 24 illustriert angegeben werden. Dieses Spektrum wird durch die folgende Fourier-Reihe repräsentiert:

$$X_n = \frac{A}{2} j(\delta(1+n) - \delta(1-n)) \tag{83}$$

Diese FR gehört zu dem Signal $x(t) = A\sin(\omega_0 t)$.

Tastet man dieses Signal mit der Periode $\triangle t$ ab und sei $k \in \mathbf{N}$ die Anzahl der Abtastwerte, so erhält man:

$$x(k\triangle t) = A\sin(\omega_0 k\triangle t + \phi) \tag{84}$$

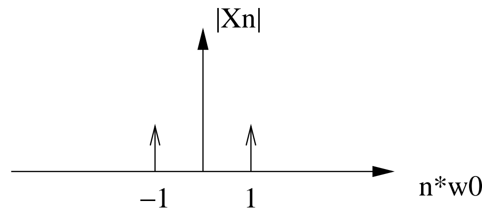


Abbildung 24: Das Amplitudenspektrum eines Sinus-Signals

Die maximale Signaländerung tritt bei den Null-Durchgängen des Sinus auf. Da man keine Aussage über ϕ machen kann, muss man eine worst-case-Abschätzung machen. Der worst-case tritt genau dann ein, wenn beide (benachbarten) Abtastwerte zeitlich gesehen gleich weit von dem Null-Durchgang des Sinus-Signals entfernt liegen. Für $\phi = 0$ liegen die Null-Durchgänge des Sinus bei $n\pi$ mit $n \in \mathbf{Z}$.

Sei nun $1/\triangle t = f_A$ die Abtastfrequenz. Dann gilt mit $t_W = 0$ für die maximale Änderung des Signals zwischen 2 Messpunkten:

$$\begin{aligned} \left|\triangle x(k\triangle t)\right|_{\max} &= A\left|\sin(-2\pi\frac{1}{2}\frac{f_0}{f_A}) - \sin(2\pi\frac{1}{2}\frac{f_0}{f_A})\right| \\ &= A\left|\sin(-\pi\frac{f_0}{f_A}) - \sin(\pi\frac{f_0}{f_A})\right| \\ &= 2A\sin(\pi\frac{f_0}{f_A}) \end{aligned} \quad (85)$$

Logischerweise geht der Grenzwert für $f_A \to \infty$ gegen Null, was nichts anderes bedeutet, als dass das Signal stetig ist.

Über das Abtasttheorem wird in diesem Zusammenhang keine Aussage gemacht - das war auch nicht das Ziel dieser Überlegungen. Ohnehin wird davon ausgegangen, dass eine Überabtastung gewährleistet ist, da nur so gewisse Aussagen bezüglich Fehlererkennung im Prozessor möglich sind.

Neben der in (85) gemachten Aussage über die maximale Signaländerung muss selbstverständlich in einem realen System der Einfluss des effektiven Messfehlers berücksichtigt werden. Sollte dieser $> 1/2$ LSB sein, so ist der entsprechende Wert für die Abschätzung dazu zu addieren. Anschließend sollte das Ergebnis aufgerundet werden.

B.3. Ein band- und amplitudenbegrenztes Signal

Der Ansatz in Anhang B.2 bringt Probleme mit sich. Im Normalfall wird die Fourier-Reihe nicht nur eine einzelne Frequenz beinhalten und es wird auch kein periodisches Signal anliegen, sodass kein Linienspektrum, sondern ein kontinuierliches Spektrum vorhanden ist. Es muss also ein Ansatz gefunden werden, der allgemeiner ist.

Oft kann man als worst case ein bandbegrenztes und amplitudenbegrenztes kontinuierliches Spektrum annehmen, wie es in Abbildung 25 prinzipiell illustriert und mit (86) angegeben ist.

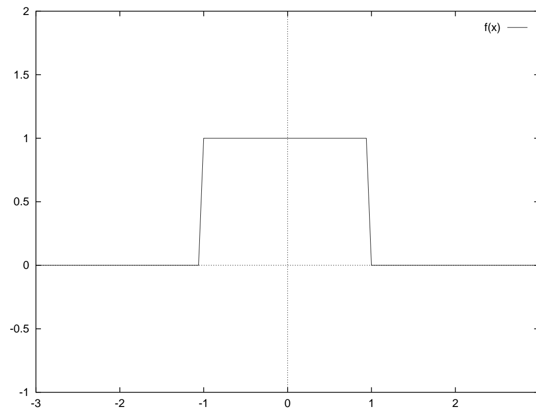


Abbildung 25: $f(x) = \mathrm{rect}(x)$

$$X(\omega) = \frac{A}{2f_0}\mathrm{rect}\Big(\frac{\omega}{\omega_0}\Big) = \begin{cases} \frac{A}{2f_0} & \text{, für } -\omega_0 \leq \omega \leq \omega_0 \\ 0 & \text{, sonst} \end{cases} \tag{86}$$

Dieses Spektrum muss man mit der inversen Fourier-Transformation in ein Zeitsignal wandeln.

$$\text{iFT:} \qquad x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(\omega)e^{j\omega t}d\omega \tag{87}$$

Damit erhält man leicht

$$x(t) = \frac{A}{\omega_0 t}\sin(\omega_o t) = A\mathrm{si}(\omega_o t) \tag{88}$$

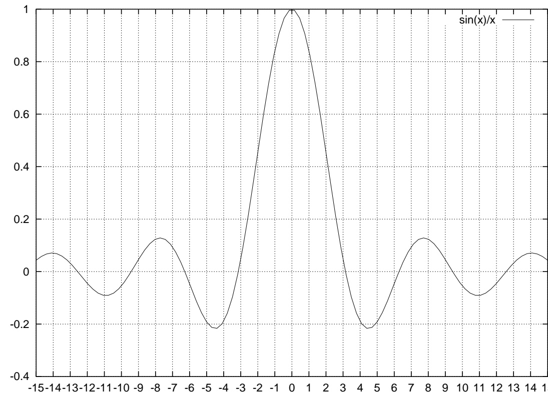


Abbildung 26: $f(x) = \mathrm{si}(x)$

Für die Bestimmung der maximalen Änderung des Signals ist die 2. Ableitung erforderlich.

$$\frac{\delta x(t)}{\delta t} = \frac{A}{\omega_0}\Big[-\frac{1}{t^2}\sin(\omega_0 t) + \frac{\omega_0}{t}\cos(\omega_0 t)\Big] \tag{89}$$

$$\frac{\delta^2 x(t)}{\delta t^2} = \frac{A}{\omega_0}\Big[\frac{2}{t^3}\sin(\omega_0 t) - 2\frac{\omega_0}{t^2}\cos(\omega_0 t) - \frac{\omega_0^2}{t}\sin(\omega_0 t)\Big] \tag{90}$$

Setzt man $\frac{\delta^2 x(t)}{\delta t^2} = 0$, so ist $t \to \pm\infty$ eine Lösung. Offensichtlich handelt es sich aber hierbei um ein Minimum der Änderung des Signals, wie man leicht an Abbildung 26 sehen kann.

Leider ist (90) nicht geschlossen lösbar. Eine Näherungslösung kann aber gefunden werden mit Hilfe des Newton-Verfahrens:

$$t_n = t_{n-1} - \frac{x''(t_{n-1})}{x'''(t_{n-1})} \tag{91}$$

Die dafür notwendige dritte Ableitung der Spaltfunktion lautet:

$$\frac{\delta^2 x(t)}{\delta t^2} = A\left[-\frac{6}{\omega t^4}\sin(\omega t) + \frac{6}{t^3}\cos(\omega t) + \frac{3\omega}{t^2}\sin(\omega t) - \frac{\omega^2}{t}\cos(\omega t)\right] \tag{92}$$

Man kann feststellen, dass das Newton-Verfahren sehr schnell konvergiert. Nimmt man beispielsweise $A = 1$ und $\omega = 1$ an, sieht man, dass schon nach der 3. Iteration ein für die Praxis ausreichendes Ergebnis ($t_{W_3} = 2{,}081575978$; keine der angegebenen Nachkommastellen ändert sich mit einer weiteren Iteration) durch das Newton-Verfahren geliefert wird.

Mit dem so errechneten Wendepunkt t_W und mit $\triangle t = \frac{1}{f_A}$ erhält man das Maximum des Anstiegs

$$\Big|x(k\triangle t)\Big|_{max} = \frac{A}{\omega_0}\Big|\frac{\sin(\omega_0(t_W + \frac{1}{2}\triangle t))}{t_W + \frac{1}{2}\triangle t} - \frac{\sin(\omega_0(t_W - \frac{1}{2}\triangle t))}{t_W - \frac{1}{2}\triangle t}\Big| \tag{93}$$

B.4. Der Einfluss von Rauschen

Gänzlich unbetrachtet blieb bis jetzt der Einfluss von Rauschen. Da Rauschen ein zufälliger Prozess ist, kann man nicht mehr ein Frequenzspektrum als worst-case - Abschätzung angeben.

Das theoretische Modell vom „weißen Rauschen“ (Leistungsdichtespektrum $S(\omega) = S_0$) erlaubt keine weitere Aussage, da die inverse Fouriertransformation angewendet auf ein konstantes Signal im Frequenzbereich zu einem Dirac-Impuls zum Zeitpunkt Null führt (als Autokorrelationsfunktion). Da dies praktisch aber nicht auftritt, kann „bandbegrenztes weißes Rauschen“ ein geeigneter Ansatz sein.

Dieser Ansatz lässt sich auch damit begründen, dass für das Leistungsdichtespektrum der Spannung über einem thermisch rauschenden Widerstand zwar $S_U(\omega) = 2kTR$ theoretisch gilt, aber oberhalb der Kreisfrequenz $hf \approx kT$ der Betrag des Leistungsdichtespektrums schnell gegen Null geht.

Sei also vereinfacht

$$S_U(\omega) = \begin{cases} S_0, & \text{wenn } |\omega| \leq \omega_G \\ 0, & \text{sonst} \end{cases} \tag{94}$$

das Leistungsdichtespektrum der zu messenden Spannung. Es gilt

$$s_U(\tau) = \frac{1}{2\pi} \int\limits_{-\infty}^{\infty} S_U(\omega) e^{-j\omega\tau} d\omega \tag{95}$$

und weiterhin

$$U_{eff} = \sqrt{\widetilde{u^2(t)}} = \sqrt{E(u^2(t))} = \sqrt{s_U(0)} \tag{96}$$

Also berechnet sich die effektive Rauschspannung zu

$$U_{eff} = \sqrt{\frac{1}{2\pi} \int\limits_{-\infty}^{\infty} S_U(\omega) d\omega} = \sqrt{\frac{S_0 \omega_G}{\pi}} \tag{97}$$

Die so errechnete effektive Rauschspannung wirkt sich additiv auf das Messsignal aus.

In einem sinnvoll dimensionierten Sensorsystem sollte nun aber der Einfluss des Rauschens so gering sein, dass eine Messungenauigkeit von maximal $\pm\frac{1}{2}$ LSB erreicht wird. Damit ergibt sich bei der Berechnung der maximalen Signaländerung lediglich die Regel, immer auf zu runden. Damit wird das Problem nicht verdrängt, sondern zu dem Design des ADC verlagert.

C. Lineare Block Codes

Im folgenden soll auf die wesentlichen Dinge, die für das Verständnis von linearen Block Codes nötig sind, eingegangen werden. Auf Beweise und tiefer gehende Erläuterungen wird verzichtet. Diese sind zu finden in [19].

C.1. Das Konzept von Block Codes

Ziel einer jeglichen Fehlerschutzcodierung ist es, robust gegenüber Übertragungsfehlern zu sein. Eine Verfälschung eines (oder mehrerer) Bits darf nicht zu einem anderen erlaubten Codewort führen. Diese Robustheit wird erreicht, indem aus einer großen Menge an Worten nur aus einer Untermenge Codewörter ausgewählt werden. Die Codewörter müssen nun so ausgewählt werden, dass jedes Codewort sich von jedem anderen möglichst stark unterscheidet. Die minimale Distanz muss möglichst groß sein. Es existiert keine Konstruktionsvorschrift für dieses Auswahlkriterium. In der Vergangenheit wurden aber einige leistungsstarke Auswahlkritierien entwickelt.

Bei einem Block-Encoder wird einem einzigen Informationsvektor U eindeutig ein einzelnes Codewort V zugeordnet. Stellt man U und V als Zeilenvektoren dar, so kann man diese Zuordnung wie folgt beschreiben:

$$\mathbf{U}\mathbf{G} = \mathbf{V} \tag{98}$$

Dabei stellt $\mathbf{G}$ die so genannte Generator-Matrix dar. Sei k die Anzahl der Informationsstellen (also die Wortbreite von U) und n; $(n \geq k)$ die Wortbreite von V, so ist $\mathbf{G}$ eine $n \times k$-Matrix.

Wenn $n = k$ kann kein Fehler erkannt werden. Dieser Fall stellt nur eine „Verwürfelung“ der Datenwörter dar.

Eine sehr gebräuchliche Form von $\mathbf{G}$ ist eine Matrix in „Standart Echelon Form“:

$$\mathbf{G}_{SEF} = (\mathbf{E}_\mathrm{k}, \mathbf{A}) = \begin{bmatrix} 1 & 0 & \dots & 0 & a_{1,1} & a_{1,2} & \dots & a_{1,(n-k)} \\ 0 & 1 & \dots & 0 & a_{2,1} & a_{2,2} & \dots & a_{2,(n-k)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 1 & a_{k,1} & a_{k,2} & \dots & a_{k,(n-k)} \end{bmatrix} \tag{99}$$

Dabei besteht $\mathbf{G}$ im ersten Teil aus der Einheitsmatrix der Größe k, was die k Informationsbits repräsentiert und im zweiten Teil aus einer Matrix $\mathbf{A}$ der Größe $(n-k) \times k$, was $(n-k)$ Prüfbits repräsentiert. Ein Code, der so beschrieben ist, wird „systematischer linearer Blockcode“ genannt.

Jede Zeile der Generatormatrix $\mathbf{G}$ stellte einen Basisvektor für den Vektorraum der Codewörter dar. Dieser Vektorraum ist ein Unterraum des n-dimensionalen Vektorraums der möglichen übertragenen Datenvektoren $\mathbf{R}$. Dabei ist $\mathbf{R} = \mathbf{V} + \mathbf{E}$ und $\mathbf{E}$ ist ein möglicher Fehlervektor.

Lineare Blockcodes besitzen eine wichtige Eigenschaft: Jeder erlaubte Codevektor ist durch Linearkombination der Basisvektoren darstellbar. Daraus folgt, dass man jede Matrix $\mathbf{G}$ in die systematische Form $\mathbf{G}_{SEF}$ überführen kann, indem man die Zeilen von

G miteinander addiert. (Allerdings gilt das nur, wenn die Elemente von **G** binäre Zahlen, also Elemente von GF(2) sind.)

C.2. Decodierung von Block-Codes - Syndrom-Decodierung

Die Codierung stellt im allgemeinen kein Problem dar. $\mathbf{UG} = \mathbf{V}$ ist leicht zu berechnen.

Nimmt man an, dass $\mathbf{E} = 0$, also $\mathbf{R} = \mathbf{V}$ und wurde mit einer systematischen Generator-Matrix $\mathbf{G}_{SEF}$ codiert, so sind die ersten k Stellen von **R** gleich **U**. Ist dagegen $\mathbf{G} \neq \mathbf{G}_{SEF}$, so kann man eine Zuordnung mittels look-up-table treffen. Es empfiehlt sich also, $\mathbf{G}_{SEF}$ zu verwenden.

Komplizierter wird es, wenn $\mathbf{E} \neq 0$, also Fehler aufgetreten sind. Es kann gezeigt werden, dass zu jedem Vektorraum, beschrieben durch **G** ein orthogonaler Vektorraum, beschrieben durch eine Matrix **H** existiert. Damit gilt

$$\mathbf{GH}^T = 0 \tag{100}$$

Dabei ist $\mathbf{H}^T$ die Transponierte der Prüfmatrix **H**. Hat man die Generatormatrix in der Form, wie in (99) beschrieben, kann man die systematische Prüfmatrix **H** direkt ableiten:

$$\mathbf{H}_{SEF} = \begin{bmatrix} a_{1,1} & \dots & a_{k,1} & 1 & 0 & \dots & 0 \\ a_{1,2} & \dots & a_{k,2} & 0 & 1 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{1,k} & \dots & a_{k,(n-k)} & 0 & 0 & \dots & 1 \end{bmatrix} \tag{101}$$

Es gilt weiterhin, dass $\mathbf{VH} = 0$ und damit

$$\mathbf{RH} = (\mathbf{V} + \mathbf{E})\mathbf{H} = \mathbf{EH} = \mathbf{S} \tag{102}$$

S wird „Syndrom“ genannt. Es ist damit klar, dass jeder Fehlervektor unabhängig vom übertragenen Codewort einem Syndrom zugeordnet werden kann. Weiterhin kann gezeigt werden, dass diese Zuordnung eineindeutig ist, wenn die minimale Distanz der Codewörter $d_{min} \geq 2d_k + 1$ ist, wobei d_k die Anzahl korrigierbarer Fehler ist und das Gewicht $w(\mathbf{E}) \leq d_k$.

Somit ist bei der Decodierung jeweils das Syndrom zu berechnen und mittels look-up-table der entsprechende Fehlervektor auszuwählen. Addiert man ihn zu **R** hinzu, erhält man $\mathbf{R} + \mathbf{E} = \mathbf{V} + \mathbf{E} + \mathbf{E} = \mathbf{V}$. Dabei kann man sich praktischerweise auf die Fehler beschränken, die in den Informationsstellen von **R** auftreten, um so die look-up-table zu verkleinern. In diesem Fall sollte man dennoch überprüfen, ob $\mathbf{E} \neq 0$, denn so kann man feststellen, dass ein Problem aufgetreten ist, auch wenn das System noch perfekt weiterarbeiten kann.

In der Literatur werden vielfältige weitere Möglichkeiten zur Decodierung von linearen Blockcodes erwähnt. Meist handelt es sich um einen Ansatz der Fehlerwahrscheinlichkeiten berechnet und somit den höchstwahrscheinlichen Datenvektor **U** aus dem Empfangsvektor **R** bestimmt. Varianten, wie „Maximum Likelihood Decoding“,

„Hard Decision Decoding“ oder auch „Soft Decision Decoding“ ist aber sehr aufwändig in der Realisierung.

Ein serieller Ansatz, der zwar bei einer Übertragungsstrecke sinnvoll ist, würde eine Taktüberhöhung erfordern beim parallelen Auslesen aus einem RAM, sodass dies ebenfalls ungünstig erscheint. „Trellis Decoding“ ist eine solche Alternative.

C.3. Zyklische Block-Codes

Mit Hilfe eines primitiven Polynoms in GF(q) kann ein zyklischer Code beschrieben werden. Sei $g(x) = g_{n-k}x^{n-k} + g_{n-k-1}x^{n-k-1} + \ldots + g_1 x + g_0$ ein solches Polynom, dann kann die Generator-Matrix des Codes wie folgt aufgestellt werden:

$$\mathbf{G} = \begin{bmatrix} g_0 & g_1 & \ldots & g_{n-k} & 0 & 0 & \ldots & 0 \\ 0 & g_0 & \ldots & \ldots & g_{n-k} & 0 & \ldots & 0 \\ \vdots & & \ddots & & & \ddots & & \vdots \\ 0 & 0 & \ldots & g_0 & g_1 & \ldots & g_{n-k} & 0 \\ 0 & 0 & \ldots & 0 & g_0 & g_1 & \ldots & g_{n-k} \end{bmatrix} \tag{103}$$

Die Matrix $\mathbf{G}$ hat n Spalten und dementsprechend $n - (n - k) + 1 = k + 1$ Zeilen.

Es sei darauf hingewiesen, dass auch bei zyklischen Block-Codes die Möglichkeit besteht, die Generator-Matrix auf die systematische Form $\mathbf{G}_{SEF}$ zu überführen, wenn die Koeffizienten $g_x \in$ GF(2).

Bekannte Vertreter der Klasse der zyklischen Block-Codes sind die BCH-Codes, sowie eine Untermenge der BCH-Codes, die Reed-Solomon-Codes.

Die spezielle Struktur der zyklischen Codes vereinfacht einige Decodieralgorithmen, die aber auf einem seriellen Ansatz basieren und daher für den Einsatz als Fehlerschutzcodierung im RAM des MSP430 schlecht geeignet sind.

C.4. Galois Felder

Ein Feld F ist eine Menge, auf der 2 Operationen definiert sind: Addition und Multiplikation. Dabei gelten folgende Axiome:

1. F bildet eines kommutative Gruppe bezüglich der Addition. Es existiert dabei ein Identitätselement bezüglich Addition, genannt „0“.

2. Die Menge der Elemtente in F ausschließlich dem Null-Element formt eine kommutative Gruppe bezüglich der Multiplikation. Es existiert ein Identitätselement bezüglich Multiplikation, genannt „1“.

3. Die Multiplikation ist distributiv über Addition. $a(b + c) = ab + ac$

Felder mit endlicher Anzahl von Elementen werden „endliche Felder“ oder auch „Galois-Felder“ genannt. Galois-Felder werden bezeichnet mit GF(q), wobei q die Ordnung des Feldes (die Anzahl der Elemente) ist.

Addition und Multiplikation in einem Galois-Feld GF(q) läuft immer modulo q ab. Von besonderem Interesse ist GF(2) für die Digitaltechnik. Die Addition kann dann als XOR-Verknüpfung und die Multiplikation als AND-Verknüpfung dargestellt werden.

Man kann nun Polynome mit Koeffizienten in GF(q) konstruieren.

$$p(x) = p_m x^m + \ldots + p_1 x + p_0 \qquad | \quad p_i \in \mathrm{GF}(q)\,;\, 0 \leq i \leq m \tag{104}$$

Sei $p(x)$ ein primitives Polynom vom Grad m und α eine primitive Wurzel von $p(x)$, so gilt $p(\alpha) = 0$. Weiterhin kann gezeigt werden, dass das durch dieses Polynom erweiterte Galois-Feld $\mathrm{F} = \mathrm{GF}(2^m)$ genau die Elemente $F = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{2^m-2}\}$ beinhaltet.

Anschaulich soll dies an GF(2^2), welches mit dem primitiven Polynom $p(x) = x^2 + x + 1$ generiert wurde, verdeutlicht werden. Es gilt $p(\alpha) = 0$, also

$$\begin{aligned} 0 &= \alpha^2 + \alpha + 1 \\ \alpha^2 &= \alpha + 1 \end{aligned} \tag{105}$$

Damit lassen sich alle Elemente von GF(2^2) wie folgt darstellen:

	Element	2-Tupel
0	0	00
1	1	10
α	α	01
α^2	$\alpha + 1$	11

Die Darstellung als 2-Tupel wurde hinzugefügt, um eine mögliche Repräsentation auf gängiger digitaler Logik zu zeigen.

D. Die Brückenschaltung

D.1. Die einfache Messbrücke

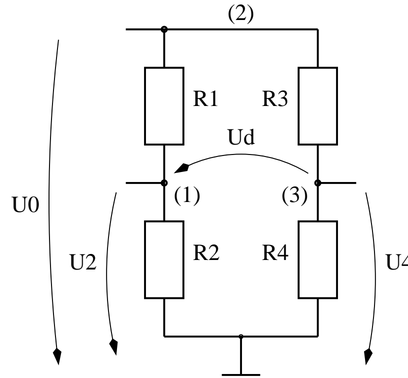


Abbildung 27: Die Brückenschaltung

Es wird von der klassischen Brückenschaltung ausgegangen. R_1 und R_4 sind drucksensitive Elemente. Sei die Brücke so abgeglichen, dass bei Normaldruck $U_d = 0$ und $U_2 = U_4 = \frac{U_0}{2}$. Die passiven Widerstände haben dann den Wert R und die sensitiven Elemente $R + \triangle R$. Dabei kann $\triangle R$ sowohl positiv als auch negativ sein. Es gilt

$$\frac{U_2}{U_0} = \frac{R_2}{R_1 + R_2} \qquad \frac{U_4}{U_0} = \frac{R_4}{R_3 + R_4} \tag{106}$$

und damit

$$\frac{U_d}{U_0} = \frac{R_4}{R_3 + R_4} - \frac{R_2}{R_1 + R_2} \tag{107}$$

Setzt man oben genannte Annahmen in die Gleichungen ein, erhält man

$$\frac{U_2}{U_0} = \frac{R}{\triangle R + 2R} \qquad \frac{U_4}{U_0} = \frac{R + \triangle R}{\triangle R + 2R} \tag{108}$$

$$\frac{U_d}{U_0} = \frac{\triangle R}{\triangle R + 2R} \tag{109}$$

Stellt man nach $\triangle R$ um, ergibt sich

$$\triangle R = R\left(\frac{U_0}{U_2} - 2\right) \qquad \triangle R = R\frac{U_0 - 2U_4}{U_4 - U_0} \tag{110}$$

$$\triangle R = 2R\frac{U_d}{U_0 - U_d} \tag{111}$$

Setzt man die Gleichungen aus (110) und (111) paarweise gleich, erhält man

$$U_0 - U_d - 2U_2 = 0 \tag{112}$$

$$U_0 + U_d - 2U_4 = 0 \tag{113}$$

$$U_2 + U_4 = U_0 \tag{114}$$

(112) und (113) bieten die Möglichkeit, auch kleinste Unterschiede zwischen den beiden Halbbrücken zu detektieren. Je nach physikalischer Erregung und nach Fehlerfall sind die Ergebnisse der linken Seiten der beiden Gleichungen größer oder kleiner Null. Eine Lokalisierung der in Kapitel 3.1 genannten Fehler ist aber nicht möglich, da es sich hier um eine 2-Sensor-Anordnung handelt.

Um eine Lokalisierung der Fehler vornehmen zu können, kann (114) verwendet werden. Darauf wird in Kapitel 3.1 näher eingegangen.

Es kann gezeigt werden, dass für eine Brückenschaltung mit 4 sensitiven Elementen die selben Ergebnisse ((112),(112) und (114)) hergeleitet werden können. Damit ist offensichtlich, dass auf diese Weise ebenso bestimmt werden kann, welche Halbbrücke einen Defekt aufweist. Will man das defekte Element genau bestimmen, so bietet die Rotation der Messbrücke (Kapitel 3.4) eine Möglichkeit dafür.

Die Annahme, dass $\triangle R$ positiv wie negativ sein darf, ist ebenfalls nicht zwingend. Auch für nur positive $\triangle R$ gelten die gefundenen Lösungen.

Ein Widerstandsabgleich nur auf $U_d = 0$ ist sehr ungünstig, da dann die Verhältnisse aller Widerstände untereinander gemessen werden müssen. Mit Hilfe dieser Verhältnisse lassen sich dann zwar auch Prüfgleichungen aufstellen, aber diese sind ungleich numerisch aufwändiger, als (112), (113) und (114).

D.2. Die dritte Halbbrücke

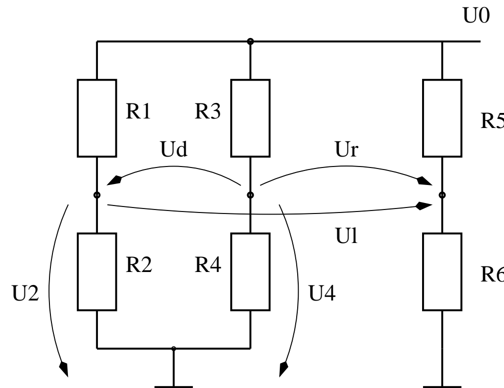


Abbildung 28: Die Brückenschaltung mit einer dritten Halbbrücke

Zur klassischen Brückenschaltung wird eine Halbbrücke hinzu genommen. Es gelten alle Annahmen, wie in Anhang D.1 und $R_5 = R_6$. Wie leicht zu erkennen ist, gilt (107) und

$$\frac{U_l}{U_0} = \frac{R_2}{R_1 + R_2} - \frac{R_6}{R_5 + R_6} \tag{115}$$

$$\frac{U_r}{U_0} = \frac{R_4}{R_3 + R_4} - \frac{R_6}{R_5 + R_6} \tag{116}$$

Setzt man alle gemachten Annahmen in die Gleichungen ein, erhält man (109) und

$$\frac{U_l}{U_0} = \frac{R}{\triangle R + 2R} - \frac{1}{2} \tag{117}$$

$$\frac{U_r}{U_0} = \frac{R + \triangle R}{\triangle R + 2R} - \frac{1}{2} \tag{118}$$

Stellt man diese Gleichungen nach $\triangle R$ um, ergibt sich (111) und

$$\triangle R = -\frac{2R}{1 + \frac{U_0}{2U_l}} \tag{119}$$

$$\triangle R = -\frac{2R}{1 - \frac{U_0}{2U_r}} \tag{120}$$

Das paarweise Gleichsetzen der 3 Terme ergibt folgende Prüfgleichungen:

$$U_d = -2U_l \tag{121}$$

$$U_d = 2U_r \tag{122}$$

$$U_r = -U_l \tag{123}$$

Die hier gefundenen Prüfgleichungen gelten ebenfalls wieder für 4 sensitive Elemente und auch für $\triangle R > 0$.

E. Abkürzungen

Viele Abkürzungen beziehen sich auf die synthesefähige Version des Microcontrollers MSP430 und sind damit auch in der technischen Dokumentation [8] aufgeführt. Zur besseren Übersicht sollen sie dennoch hier noch einmal wiedergegeben werden.

ADC	analog to digital converter
BCD	binary coded decimals - Dezimalzahlen, codiert mit 4 Bit Binärcode pro Dezimalziffer; Addition wie in [15]
CG1	constant generator 1 = gleichzeitig →SR
CG2	constant generator 2 = Register R3
DSP	digital signal processor
ECC	error correction code
IE	interrupt enable - Siehe dazu auch →SFR.
IFG	interrupt flag - Teil der →SFR.
IRQ	interrupt request
IRQA	→IRQ accepted
ISR	interrupt service routine
LPM	low power modus - siehe [7], Kapitel 3
LSB	least significant bit(s)
MAB	main address bus
MDB	main data bus - Besteht aus 2 Teilen: MDB_in und MDB_out, da nicht bidirektional.
MSB	most significant bit(s)
PC	program counter = Register R0 - Zeiger auf den aktuellen Befehl im RAM.
POR	power-on reset →[7], Kapitel 3
POST	power-on self test
PUC	powered up clear →[7], Kapitel 3
RESO	recomputing with shifted operands
REiSO	recomputing with invers shifted operands
RISC	reduced information set computer
RNS	residue number system
RSD	redundant signed digit correction
SEF	standart echelon form
SFR	special function register - Bereich des RAM von 00h bis 0Fh; byte access (siehe [7], Kapitel 3)
SP	stack pointer = Register R1 - Zeiger auf das oberste Element im stack (→TOS). (Anmerkung: Der stack wird in Richtung kleinerer Adressen gefüllt.)
SR	status register = Register R2
TOS	top of stack - das Element im RAM auf den der →SP zeigt.
WDTCTL	watchdog/timer control - RAM-Bereich bei 0120h zur Steuerung diverser Reset-Eingenschaften des MSP430 (siehe [7], Kapitel 3 und 10)

Literatur

[1] Dirk Weiler: Dissertation „Selbsttest und Fehlertoleranz mit zugelassener milder Degradation in integrierten CMOS-Sensorsystemen“; Gerhard-Mercantor-Universität - Gesamthochschule Duisburg; 7. Juni 2001

[2] Christian Mayr: Präsentation im Rahmen des Oberseminars „VLSI-Schaltungen und Systeme“; 12.07.2001; TU Dresden; Dozent: Prof. Schüffny

[3] J. Schreiter, A. Graupner, S. Getzlaff, R. Srowik, R. Schüffny: „A CMOS Image Sensor with Parallel Analog Processing Unit for Transformations and Spatial Convolutions“; TU Dresden;
http://www.iee.et.tu-dresden.de/~hpsnweb/hdown//sci2000.pdf

[4] Hidetoshi Onodera, Tetsuo Tateishi, Keikichi Tamaru: „A Cyclic A/D Converter That Does Not Require Ratio-Matched Components“;
IEEE Journal Of solid-State Circuits, vol. 23, no. 1, February 1988

[5] Michael Fischell, Stephan Latzel, Walter Anheier: „Online/Offline Tests für integrierte Sensoren in der Betriebsphase“
http://www.item.uni-bremen.de/research/papers/
paper.pdf/Michael.Fischell/itg01/itg01.pdf

[6] Mandeep Singh, Israel Koren: „Incorporating Fault Tolerance in Analog-to-Digital Converters (ADCs)“;
http://euler.ecs.umass.edu/research/siko02.pdf
„Reliabilty Enhancement of Analog-to-Digital Converters“;
http://euler.ecs.umass.edu/research/siko01.ps

[7] MSP430 User's Guide
http://www.ti.com/sc/docs/products/micro/msp430/docs.htm

[8] Ralf Hildebrandt: „Synthesefähige Beschreibung des Microcontrolers MSP430“; noch unveröffentlichte technische Dokumentation; Fraunhofer IMS, Dresden

[9] John Lach, William H. Mangione-Smith, Miodrag Potkonjak: „Low Overhead Fault-Tolerant FPGA Systems“;
http://www.klabs.org/richcontent/Papers/LowOverheadFaultTolerance.pdf

[10] Neil W. Bergmann, Anwar S. Dawood: „Reconfigurable Computers in Space: Problems, Solutions and Future Directions“;
http://klabs.org/richcontent/MAPLDCon99/Papers/P2_Bergmann_P.PDF

[11] D. W. Caldwell, D. A. Rennels: „Minimalist Fault Masking, Detection and Recovery Techniques for Mitigating Single Event Effects in Spaceborne Microcontrolers“;
http://www.cs.ucla.edu/~rennels/tr9825dc.pdf
Weitere Details: http://www.cs.ucla.edu/~rennels/dougdiss.pdf

[12] Janak H. Patel: „RESO - Recomputing with Shifted Operands“; http://courses.ece.uiuc.edu/ece442/lecture_10.pdf

[13] Dr. Zbigniew Kalbarczyk: „Design of Reliable Systems and Networks: ECE 442 / CS 436 Lecture 8 - Information Redundancy Coding“ http://courses.ece.uiuc.edu/ece442/lecture_8.pdf

[14] J.C. Muzio, M. Serra: „H. Codes“ http://www.csc.uvic.ca/˜csc454/notes/H.Codes.pdf

[15] Prinzip der BCD-Addition: http://ee.ntu.edu.au/staff/saeid/teaching/tbe254/lectures/cldch05/sld038.htm

[16] Markus Wannemacher, Reiner Lichtenecker, Wolfgang A. Halang: „Entwurfsmethode für einen universellen Koprozessor für zeitkritische Aufgaben in sicherheitsgerichteten Echtzeitsystemen“ http://www.fernuni-hagen.de/IT/papers/pb98.ps.gz

[17] K.S. Papadomanolakis, A.P. Kakarountas, V. Kokkinos, N. Sklavos, C.E. Goutis: „The Effect of Fault Secureness in Low Power Multiplier Designs“ http://patmos2001.eivd.ch/program/Repro%5CArt_10_3.pdf

[18] Ismet Bayraktaroglu, Alex Orailoglu: „Accumulation-Based Concurrent Fault Detection for Linear Digital State Variable Systems“ http://www.cs.ucsd.edu/˜ibayrakt/publications/ats00.pdf

[19] L. H. Charles Lee: „Error-Control Block Codes“; ISBN 1-58053-032-X

[20] G.A. Jullien: „Number Theoretic Techniques in Digital Signal Processing“; http://wooster.hut.fi/geta/courses/graham/Handouts/NumberTheory.pdf

[21] T.H. Liew, L-L. Yang, L. Hanzo: „Soft-Decision Redundant Residue Number System Based Error Correction Coding“; http://www-mobile.ecs.soton.ac.uk/thl97r/papers/paper-rev2-web.pdf

[22] Robin Chapman: „A Guide to Arithmetic“; http://www.maths.ex.ac.uk/˜rjc/notes/arith.dvi

[23] Reto Zimmermann: „Computer Arithmetic: Principles, Architectures and VLSI Design“ (Lecture notes) http://www.ece.cmu.edu/˜ece347/handouts/comp_arith_notes.pdf

[24] M.G. Parker, M. Benaissa: „Fault-Tolerant Linear Convolution Using Residue Number Systems“; http://www.ii.uib.no/˜matthew/fltprns.pdf

[25] Prof. Doran Wilde: „Computer Arithmetic“ (Lecture 4); http://www.ee.byu.edu/ee/class/ee621/Lectures/L04.PDF

[26] „The Basics of Current Loop“;
http://www.bb-elec.com/tech_articles/current_loop_app_note/basics_of_current_loop.asp

[27] Siemens Moore 345 XTC;
http://www.moore-solutions.com/instrumentation/PDF's/XTC%20Critical.pdf
http://www.sea.siemens.com/instrbu/docs/pdf/adql-6r1.pdf

[28] SAAC - ECSW Series;
http://www.ssac.com/cgi-bin/compile_html.pl/wall/data/sect-h/ecsw/ecswdata.htm

[29] Erfaan Sharif, Tony Dorey, Andrew Richardson: „An Integrated Diagnostic Reconfiguration Technique for Fault Tolerant Mixed Signal Microsystems“
http://www.comp.lancs.ac.uk/microsystems/publications/icecs_idr_98.pdf

[30] C. Jeffrey, R. Rosing, A. Richardson: „A Built-in Test Solution for a SMART Silicon Micromachined Resonant Pressure Sensor“;
http://cezanne.inesc.pt/etw00/Program/SessionP1/etwp1_1.pdf

[31] Hayley Iben, Ali Lakhia, Rachel Rubin: „Watchdog Designs for TinyOS Motes“;
http://www.cs.berkeley.edu/~iben/cs252/

[32] Harris 80RH86 in einem Satellit
http://www.ll.mit.edu/ST/sbv/app1_hardware.html

[33] 32 bit intel hex format .hex
http://www.pjrc.com/tech/8051/pm2_docs/intel-hex.html

[34] Peter König, Mitarbeiter Fraunhofer IMS Dresden, Betreuer dieser Arbeit

Es sei darauf hingewiesen, dass aufgrund der Schnelllebigkeit des Internets aufgeführte Links eventuell im Laufe der Zeit nicht mehr gültig sind. In diesem Fall wird empfohlen Suchmaschinen, wie http://www.google.de oder http://citeseer.nj.nec.com/cs zu benutzen, um die Dokumente wieder aufzufinden.