IMPRS course

Scientific Toolbox : Data Acquisition

Template: "Vorlesungsbegleitendes Skript, PDV1,
Technische Fachhochschule Berlin vom 19.4.06"

Fritz-Haber-Institut der Max-Planck-Gesellschaft
Heinz Junkes (junkes@fhi-berlin.mpg.de)

In 2006 I prepared this script for my course PDV (Prozessdatenverarbeitung) at the Technical University of Applied Sciences (now named Beuth Hochschule für Technik Berlin), I didn't pursued this topic for longer.

For my first IMPRS course I made an update and translation of some existing chapters into English. Unfortunately I had no time to revise everything again, and therefore some inconsistencies in the script may be found. Please consider it as a first draft. This version is on progress and I am grateful for all comments and suggestions.

Inhaltsverzeichnis

1 PDV 1 Themenübersicht

In dem Kapitel **"Was heißt Prozessautomatisierung"** wird als erstes erläutert, was man unter einem Realzeitsystem versteht und welche Bedeutung der Automatiserungsgrad hat. Es werden unterschiedliche Rechner-Einsatzarten aufgezeigt und die Unterscheidung zwischen Produktautomatisierung und Anlagenautomatisierung vorgenommen. Die Bestandteile eines Automatisierungssystems werden vorgestellt und dabei die unterschiedlichen Ebenen und ihre Anforderungen aufgezeigt. Danach werden einzelne Vorgänge klassifiziert und Darstellungsarten für Automatisierungssysteme besprochen.

In dem Kapitel **"Automatisierungsgerätesysteme und -strukturen "** werden die unterschiedlichen Automatisierungscomputer vorgestellt und die besondere Arbeitsweise einer Speicherprogrammierbaren Steuerung (SPS) dargestellt. Es wird zwischen zentralen und dezentralen Strukturen unterschieden. Es werden Kombinationen von Automatisierungsstrukturen, Automatisierungshierarchien und deren Anforderungen aufgezeigt. Dezentrale Automatisierungssysteme und die Grundstrukturen der Kommunikation werden untersucht. Es wird zwischen offenem und einem geschlossenem Kommunikationssystem unterschieden. Der Begriff der Redundanz wird mit den verschiedenen Arten von Hardware-Redundanz vorgestellt und die Bedeutung von (Software-) Diversität erläutert.

Das Kapitel **"Prozessperipherie "** behandelt die Schnittstellen in einem Automatisierungssystem. Stellt Sensoren und Aktoren vor und und wie diese aufgebaut werden. Die Datendarstellung in Automatisierungscomputern wird besprochen und wie eine Analog-Digital-Umsetzung und Digital-Analog-Umsetzung durchgeführt werden kann. Unterschiedliche Umsetzer-Realisierungsformen werden aufgezeigt, sowie die Realisierungsmöglichkeiten digitaler Ein- und Ausgabe von Prozesssignalen. Feldbussysteme werden mit ihren unterschiedlichen Bus-Zugriffsverfahren erklärt und hierbei besonders die PROFIBUS-Kommunikation und die wichtigsten Eigenschaften von CAN und FlexRay.

Im Kapitel **"Echtzeitprogrammierung "** werden die Forderungen bei der Echtzeit-Programmierung vorgestellt und zwischen harter und weicher Echtzeit unterschieden. Die synchrone Programmierung sowie die asynchrone Programmierung werden aufgezeigt. Es werden Rechenprozesse vorgestellt und welche Möglichkeiten es zur zeitlichen Synchronisierung (z.B. Semaphorvariable) gibt. Unterschiedliche Scheduling-Verfahren werden angewendet und ein Schedulability-Test vorgstellt.

Im Kapitel **"Echtzeitbetriebssysteme "** wird erkärt, was man unter Betriebsmitteln versteht. Die Aufgaben eines Echtzeit-Betriebssystems werden vorgestellt. Interrupts und die Speicherverwaltung werden erläutert. Das Vorgehen bei der Entwicklung eines Mini-Betriebssystems und der Aufbau eines Beispiel-Mini-Betriebssystems werden dargestellt. Eine Übersicht über Echtzeit-Betriebssysteme beendet das Kapitel.

Im Kapitel **"Programmiersprachen für die Prozessautomatisierung "** wird die Vorgehensweise bei der Erstellung von Prozessautomatisierungsprogrammen dargestellt. Programmiersprachen werden nach Sprachhöhe unterschieden und spezielle Programmiersprachen für SPSen werden besprochen (mit einfachen Beispiele in SPS-Sprachen). Die wichtigsten Echtzeit-Konzepte für Programmiersprachen werden aufgezeigt und die Echtzeit-Konzepte in Ada 95 (Entwurf Echtzeit-Programm in Ada 95). Es wird C/C++ bezüglich Echtzeit eingeschätzt und vorgestellt

worauf die Portabilität von Java beruht. Es werden die Echtzeit-Erweiterungen von Java vorgestellt.

In the chapter **"Data acquisition software/frameworks "** some data acquisition frameworks are described. Among them are two commercial (MATLAB and LABView) and some freely available. It tries to identify the areas of application and features of the different software packages. The focus is on EPICS and Control System Studio.

2 Was heißt Prozessautomatisierung

2.1 Definition einiger Grundbegriffe

2.1.1 Technischer Prozess

Definition Prozess / Technischer Prozess (DIN 66201):

Ein Prozess ist eine Gesamtheit von aufeinander einwirkenden Vorgängen in einem System, durch die Materie, Energie oder Information umgeformt oder gespeichert werden. Ein technischer Prozess ist ein Prozess, dessen physikalische Größen mit technischen Mitteln erfasst und beeinflusst werden.

Ein technischer Prozess ist ein Vorgang, durch den Materie, Energie oder Information in ihrem Zustand verändert wird. Diese Zustandsänderung kann beinhalten, dass ein Anfangszustand in einen Endzustand überführt wird.

Systeme bestehen aus:

- Module
- Schnittstellen
  - physikalisch
  - energetisch
  - informationell

Prozessbegriff beschreibt

System + **Zeit**

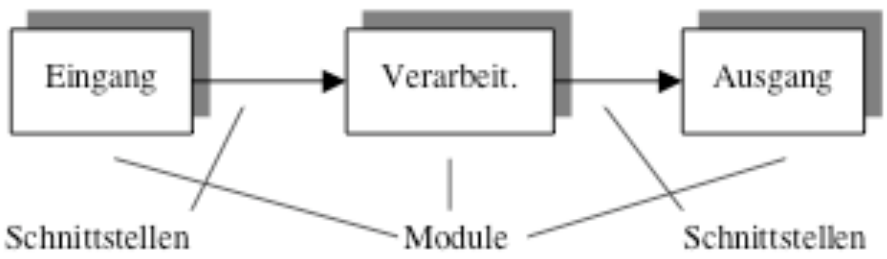


Abbildung 2.1: Systeme

Anfangszustand	**Technischer Prozess in einem technischen System**	**Endzustand**
niedrige Raumtemperatur	Wärmevorgänge bei der Beheizung eines Wohnhauses mit einer Ölheizungsanlage	erhöhte Raumtemperatur
verschmutzte Wäsche	Waschvorgang in einer Waschmaschine	saubere Wäsche
unsortierte Pakete	Transport- und Verteilvorgänge bei einer Paketverteilanlage	nach Zielorten sortierte Pakete
fossile oder Kernbrennstoffe	Energie-Umwandlungs- und Erzeugungsvorgänge in einem Kraftwerk	elektrischer Strom
einzulagernde Teile	Lagervorgänge in einem Hochregallager	zu Kommissionen zusammengestellte Teile
Zug in Ort A	Verkehrsablauf bei der Fahrt eines Zuges	Zug in Ort B
monomerer Stoff	Vorgänge in einem chemischen Reaktor	polymerer Stoff
ungeprüftes Gerät	Prüfabläufe in einem Prüffeld	geprüftes Gerät
Teile ohne Bohrung	Bohrvorgang bei einer Bohrmaschine	Teile mit Bohrung
Schadstoffe in der Luft	Vorgänge in einem System zur Schadstoffüberwachung der Luft	Informationen über Schadstoffkonzentrationen werden in der Überwachungszentrale angezeigt

Tabelle 2.1: Beispiele technischer Prozesse

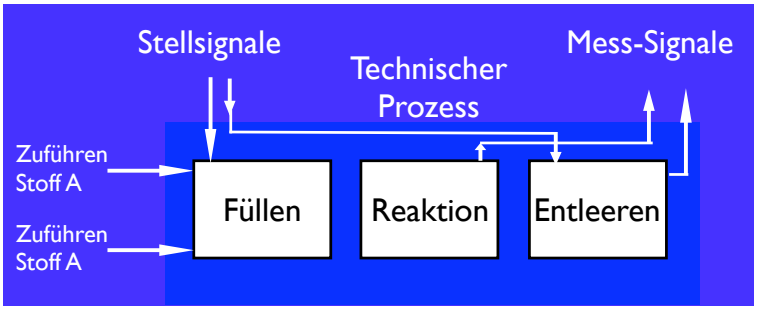


Abbildung 2.2: Beispiel: chemischer Reaktor

Abbildung 2.3: Bedienoberfläche (chemischer Reaktor)

Technische Prozesse können sehr einfach wie z.B. Waschmaschinen oder aber auch sehr komplex wie z.B. bei Walzwerken sein. Wobei unterschiedliche Teilprozesse zu einem Gesamtprozess vereinigt werden können wie z.B. in Autos mit der KFZ-Elektronik.

Der technische Prozess läuft auf einem technischem System ab.

Beschreibung eines Prozesses

- Prozess (physikalische Größen)
  - Prozesskennwerte, (Spezifikation) Grenzwerte
  - Prozessparameter (aktuelle Einstellung)
  - Prozesszustand (aktueller Zustand)
- Schnittstelle zur Umgebung
  - Eingangsgrößen
  - Stellgrößen

	Transport	**Umformung**	**Speicherung**
Materie	Handhabung, Straße, Schiene	Bearbeitung, Verbindung, Fügen	Lager
Energie	EV, Hochspannungstechnik	Transformation, Elektrochemie	Batterie, Stauwerk
Information	Telekommunikation	Rechnertechnik	Speicher

Tabelle 2.2: Beispiel Prozess-Klassen

  - Störungen
  - Ausgangsgrößen (Alarme)

Einige Größen sind schwierig messbar: z.B. Rauhigkeit, Textur einer Oberfläche (keine einheitliche Messvorschrift)

Klassifikation von Prozessen

- Nach **Verarbeitungsart** (was wird gemacht?)
  - Transport
  - Umformung
  - Speicherung
  - (Typische Ketten)
- Nach **Verarbeitungsgut** (was wird bearbeitet?)
  - Materie
  - Fertigung (geometrische Form), Maschinenbau, Optik, Elektrotechnik
  - Verarbeitung (Form allgemein: vorwiegend nicht Metall) Papier, Textil...
  - Verfahrensprozess (Form gleichgültig) chemische Industrie, Metallurgie
  - Energie
  - Information
- nach **Vorhersehbarkeit**
  - deterministischer Prozess (mit bzw. ohne Störgrößen)
  - stochastischer Prozess
- nach **Komplexität** (Handhabung der Komplexität?)
  - Elementarprozess

    1 Verarbeitungsart & 1 Verarbeitungsgut
  - Einzelprozess

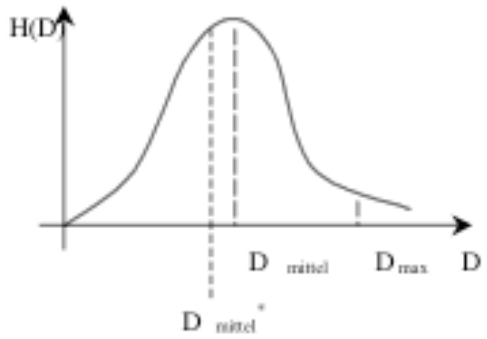


Abbildung 2.4: Durchsatz

kleinste geschlossene Prozesseinheit, die einen kompl. Arbeitsablauf erledigt, z.B. Waschmaschine

- Verbundprozess

  umfasst Einzelprozesse, die zusammenarbeiten, um eine gemeinsame Aufgabe zu erledigen $\Rightarrow$ Fertigungsstraße (z.B. Automobilbau)

- Betriebsprozess

  Umfasst verschiedene Arbeitsbereiche eines Betriebes

- nach zeitlichem Ablauf
  - kontinuierlich (Fließprozess)
  - diskontinuierlich: in Chargen, diskret oder Stückprozess
  - gleichmäßig bzw. ungleichmäßig

wichtige Prozesskennwerte:

- allgemein:
  - Durchsatz
  - Auslastung

$$Auslastung = \frac{D_{mittel}}{D_{max}} = 80 - 95\% \tag{2.1}$$

- bei Flussprozessen
  - Durchsatz (Menge / Zeit)
  - Gleichmäßigkeit (fester / variabler Takt)
  - Taktzeiten

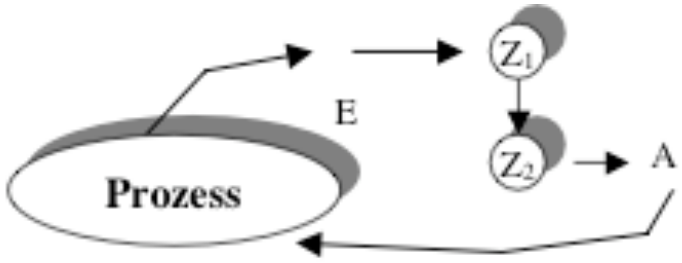


Abbildung 2.5: Zustandsdiagram Automat

- bei Chargen
  - Chargengröße
  - Chargentaktzeit

2.1.2 Prozessautomatisierung

Der Begriff Automatisierung leitet sich aus dem griechischen Wort *automatos* ab, das mit "sich selbst bewegend" oder "aus eigenem Antrieb" übersetzt werden kann. Im technischen Sinne bedeutet Automatisierung, anders als im umgangssprachlichen, immer einen Einsatz von Maschinen und nicht das Handeln ohne Nachzudenken, wie es in der Sportpsychologie verwendet wird.

Prozessautomatisierung bedeutet, wie der Begriff es auch schon bezeichnet, das Automatisieren eines Prozesses. Dazu wird ein Automat benötigt. Der Automat ist ein selbsttätig arbeitendes technisches System. Es kann sich dabei um einen einfachen Automaten, wie den berühmten Kaugummiautomat oder auch Zigarettenautomat handeln, aber auch um komplexere Systeme wie den Fahrkartenautomaten bei der Deutschen Bahn oder BVG.

Definition Automat (DIN 19233):
Ein Automat ist ein künstliches System, das selbständig einem Programm folgt. Auf Grund des Programms trifft das System Entscheidungen, die auf Verknüpfungen des Systems beruhen und Ausgaben zur Folge haben.

Von Automatisierung spricht man, wenn man ganze Anlagen mit ihren Maschinen in die Lage versetzt selbständig zu arbeiten. Dies ist z.B. bei der Büroautomatisierung, Verkehrsautomatisierung, Bahnautomatisierung und Prozessautomatisierung der Fall. Dabei ist die Prozessautomatisierung die Automatisierung technischer Prozesse. Unter einem Automatisierungssystem versteht man die Zusammenschaltung eines technischen Systems mit den dazu benötigten Rechner- und Kommunikationssystemen sowie dem Prozessbedienpersonal.

Die Zielvorstellung ist die Automatisierung der Vorgänge des technischen Prozesses mit Hilfe von entsprechenden Informationsverarbeitungseinheiten. Der Mensch gibt nur noch die Wünsche an das Betriebsergebnis vor.

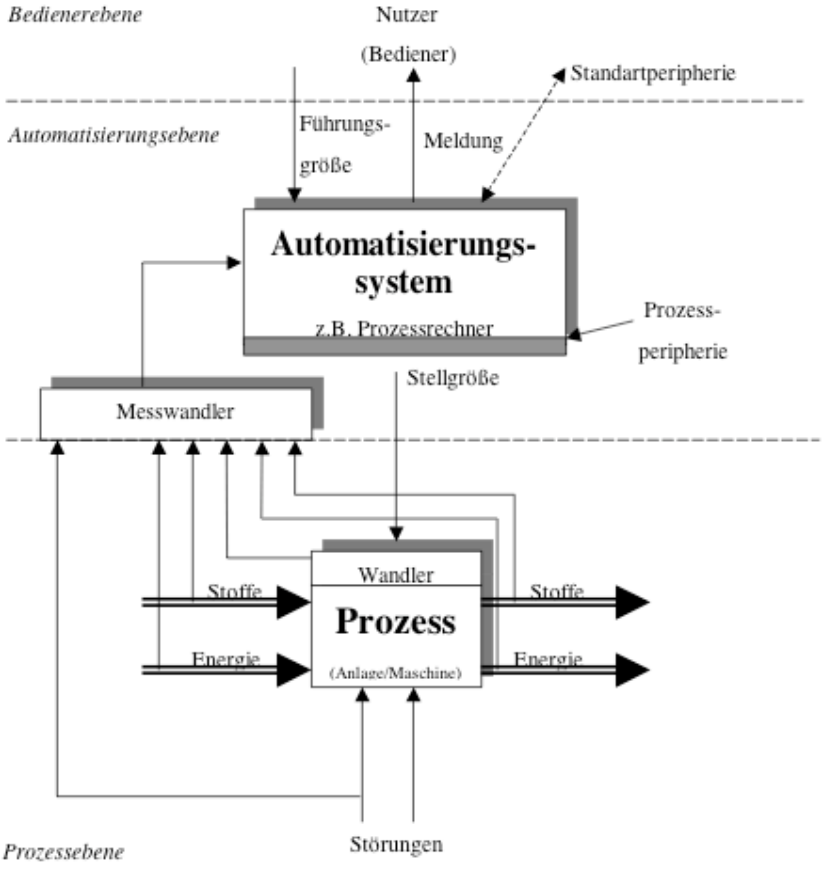


Abbildung 2.6: Automatisiertes System

2.1.3 Ziele der Automatisierung

Im unternehmerischen Bereich lassen sich Ziele in die Bereiche der strategischen und der operativen Ziele einteilen. Das Hauptziel im strategischen Bereich ist die langfristige Sicherung des Unternehmens. Wenn dieses Ziel weiter aufgegliedert wird, lassen sich Unterziele identifizieren, die als Triebkräfte für die Automatisierung gesehen werden können:

- Bessere und konstantere Qualität,
- Arbeitskräftemangel (beispielsweise in Japan und jetzt auch in Deutschland) und
- zu hohe Lohnkosten.

Die operativen Ziele, die bei der Automatisierung verfolgt werden sind Rationalisierung, Qualitätsverbesserung, Humanisierung der Arbeit, Ersatz- und Erweiterungsinvestitionen.

2.1.4 Grenzen der Automatisierung

Wenn ein Fertigungssystem automatisiert wird, stellt sich die Frage, in wie weit dieses möglich ist. Grundsätzlich liegt diese Grenze zunächst bei 100%, also eine vollständige Automatisierung, in der der Mensch keine Aufgaben mehr übernimmt. Es gibt aber noch andere, weichere, Grenzen:

Technische Grenzen

Durch die Entwicklung der Technik in den letzten zehn Jahren sind der Automatisierung aus technischer Sicht kaum noch Grenzen gesetzt. Ein deutliches Zeichen hierfür ist, dass die meisten Entscheidungsträger sich nicht durch die technischen Möglichkeiten bei der Automatisierung beschränkt fühlen. Die Grenzen liegen vielmehr in der Identifizierung der zu bewältigenden Aufgaben. Ein Hindernis scheint die Geschwindigkeit, mit der Daten technisch verarbeitet werden können, zu sein. Die Menge von Informationen kann eine Maschine überfordern, da sie nicht über dieselbe Selektionsmöglichkeit wie ein Mensch verfügt. Außerdem ist es schwer, Maschinen so weit zu bringen, dass sie lernfähig werden.

Gesellschaftliche Grenzen

Rechtliche Grenzen

Wenn Maschinen Aufgaben übernehmen, die vormals von Menschen übernommen wurden, gibt es grundsätzlich keine Probleme, wenn dadurch der Arbeitplatz angenehmer wird. Anders ist es aber, wenn Maschinen Kontrollaufgaben übernehmen sollen oder selbstständig entscheiden. Wie z.B. in der DIN 11 489 für die Milchwirtschaft beschrieben, steigen die rechtlichen Anforderungen mit dem Automatisierungsgrad.

Grenzen in der Akzeptanz

Eine weitere Hemmnis bei der Automatisierung ist die Akzeptanz der Betroffenen und der Entscheidungsträger. Betroffene müssen sich darauf einlassen, dass Maschinen ihnen Arbeiten abnehmen und sich auch auf diese verlassen. Wenn ein Mitarbeiter seine Arbeitszeit verwendet, um eine Maschine zu kontrollieren, die keine Kontrolle benötigt, entstehen Kosten, die nicht erwünscht sind. Die Akzeptanz der Entscheidungsträger ist notwendig, um ein Automatisierungsprojekt umsetzen zu können. Sie müssen also vom Nutzen der Veränderung überzeugt werden. (siehe auch ökonomische Grenzen)

Soziale Grenzen

Durch die fortschreitende Automatisierung treten Veränderungen in der Gesellschaft auf. Es werden beispielsweise weniger Arbeitskräfte benötigt, um ein Produkt zu erzeugen. Diese Entwicklung muss von der Gesellschaft akzeptiert werden, da sie auch durch sie getragen wird. Gemäß Henning ist ein Automatisierungsingenieur nicht nur für die Sicherheit der Anlage, sondern auch für die Auswirkungen auf die Benutzer verantwortlich. [12] Dies bedeutet, dass auch die Auswirkungen bezüglich der veränderten Arbeitsanforderung bedacht werden müssen.

Ökonomische Grenzen

Die Grenze, an der die meisten Automatisierungsprojekte scheitern, liegt im ökonomischen Sektor. Die Automatisierung soll bewirken, dass Prozesse effizienter durchgeführt werden können. Dies geschieht meist durch den Ersatz humaner Ressourcen durch Ressourcen im technischen Bereich. Eine wichtige Maßzahl hierfür ist die Amortisationszeit. Sie drückt aus, nach welcher Zeitspanne sich die Investition in eine neue Anlage im Vergleich zur bestehenden Produktion rechnet. Laut einer Studie stimmen Entscheidungsträger einem Automatisierungsprojekt nur bei einer Automatisierungsdauer zwischen einem halben und fünf Jahren, je nach Branche, zu. [14]

2.1.5 Prozessleittechnik

Prozessleittechnik unterscheidet sich von der Automatisierungstechnik dadurch, dass bei ihr die Gesamtheit aller Aufgaben (Funktionen) betrachtet wird, die für die Steuerung eines Prozesses erforderlich ist, und nicht nur die Untermenge derer, die über die jeweilige Automatik zu einem Stelleingriff führen. Insbesondere werden also neben den Funktionen der jeweiligen Automatik, die in deren Programm festgelegt sind, auch alle Entscheidungsprozesse des Menschen, der den jeweiligen Prozess überwacht, und seine daraus resultierenden unmittelbaren Eingriffe in den Prozess mit berücksichtigt.

Aus kybernetischer Sicht werden also nicht nur das im Rechner gespeicherte, sondern auch das im Gehirn des Operators oder des leitenden Ingenieurs vorhandene Prozessmodell für die Prozesssteuerung herangezogen. Insofern ist Prozessleittechnik inhaltlich umfassender als Automatisierungstechnik.

Auf der anderen Seite werden auch in der Automatisierungstechnik zunehmend Fragen der Mensch-Prozess-Kommunikation behandelt, wobei ergonomische Aspekte eine wesentliche Rolle spielen. Damit werden aber ebenfalls Fähigkeiten des den Prozess leitenden Menschen

angesprochen, unter anderem auch seine mentalen Prozesse. Hierdurch verwischt sich der Unterscheid zwischen den beiden genannten Fachgebieten etwas.

2.1.6 Prozessinformatik

Hierbei steht die Automatisierungssoftware mit dem Rechner- und Kommunikationssystem im Vordergrund.

Die theoretische Informatik beschäftigt sich mit den begrifflichen Grundlagen der Informatik und insbesondere mit ihrer Einbettung in die Logik und die Mathematik, mit der maschinellen Berechenbarkeit von Problemen und mit der Effizienz von Algorithmen. Aus Sicht der Prozessinformatik dominieren jedoch die Anwendbarkeit und die Anwendung von Rechnern für deren Problemstellungen.

Ganz anders sieht dies bei der technischen und praktischen Informatik aus. Setzt man, wie weit verbreitet,

- Technische Informatik = Rechner-Hardware,
- Praktische Informatik = Rechner-Software,

so erweisen sich diese beiden Aspekte als immens wichtig für die Prozessinformatik. Die Anforderungen der Automatisierungstechnik haben weitreichende Auswirkungen auf Hardware und Software der Rechner, die für die Steuerung und Überwachung technischer Prozesse eingesetzt werden (Prozessrechner), sowie auf die der darauf aufbauenden Automatisierungssysteme.

2.1.7 Echtzeitsystem (Realzeitsystem)

In der DIN wird der Begriff Realzeitsystem benutzt.

Definition Realzeitbetrieb (DIN 44300):

Ein Betrieb eines Rechensystems, bei dem Programme zur Bearbeitung anfallender Daten ständig betriebsbereit sind derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu bestimmten Zeitpunkten anfallen.

Realzeitbedingungen:

- Direkte Kopplung mit der Welt
- Schritthaltende Verarbeitung
- Leistungssteigerung
- Sicherheitssteigerung

Anforderungen an Echtzeitsystem:

- **Rechtzeitigkeit**, zur richtigen Zeit reagieren, nicht zu früh, nicht zu spät
- **Gleichzeitigkeit**, auf mehrere Dinge gleichzeitig reagieren, parallele Abläufe

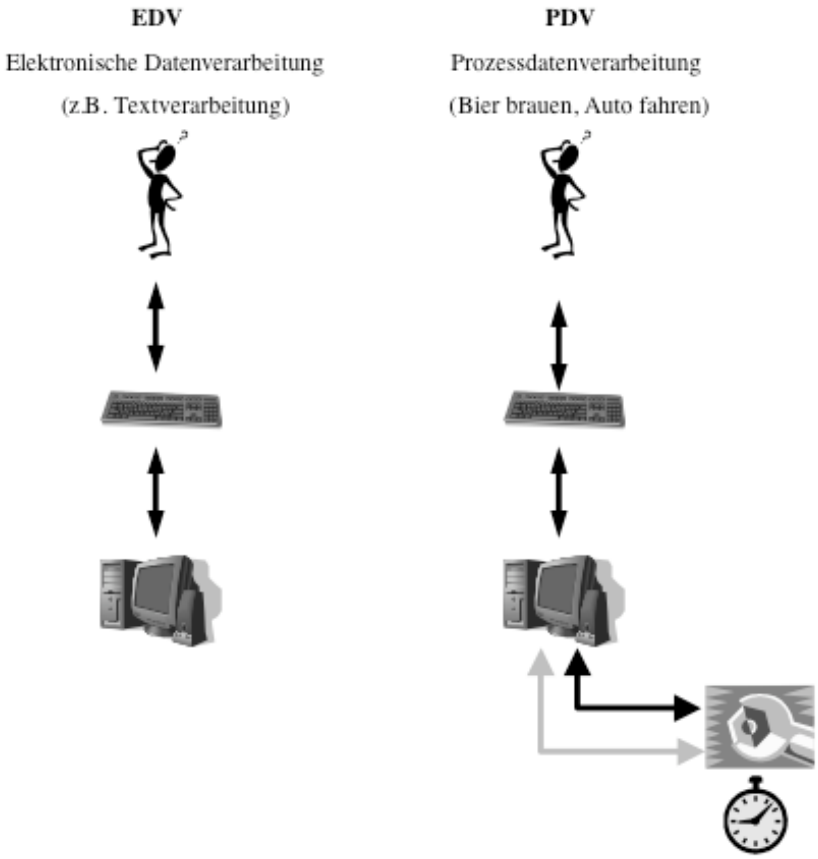


Abbildung 2.7: EDV vs. PDV

- **Verlässligkeit**, zuverlässig, sicher, verfügbar (wichtiger Kaufgrund)
- **Vorhersehbarkeit**, alle Reaktionen müssen planbar und deterministisch sein, nachvollziehbar im Fehlerfall

2.1.8 Automatisierungsrechner

Der Prozessrechner ist mittels Prozessperipherie direkt an den Prozess gekoppelt. Jedes Rechnersystem ist geeignet, welches die Anforderungen des Prozesses erfüllt. Dazu gehören:

- Realzeitfähigkeit. Der Rechner muß auf Ereignisse (Alarme) im Prozess hinreichend schnell reagieren (Mikrosekunden bis Stunden).
- Möglichkeit zum Setzen von Prioritäten beim Ablauf.
- Speichergröße, um die Datenmengen verlustlos zu bewältigen.
- Verarbeitung von Zahlen, Zeichen und Bits
- Geeignete Schnittstellen.

In 60er und 70er Jahren gab es spezielle Prozessrechner (z.B. DEC PDP-8/11). Dieser Begriff ist mittlerweile veraltet, da die Unterschiede zu den *normalen* Rechnern verschwunden sind.

2.1.9 Automatisierungsgrad

Der Automatisierungsgrad stellt einen Wert dar, der zeigen soll, in welchem Ausmaß ein Prozess automatisiert abläuft. Dieser Wert steht als *Kennziffer für die Planung von Fertigungssystemen*. Aus ihm soll also eine Vorgabe für die Planung entwickelt werden. Um aus einem System die höchste Leistung zu gewinnen, ist es notwendig, einen Teil der zu bewältigenden Funktionen zu automatisieren. Interessant ist nun der optimale Grad der Automatisierung, um die Effizienz eines Prozesses zu maximieren. Dieser Grad kann, wenn überhaupt, nur für den Einzelfall beantwortet werden.

Wie schon bei der Automatisierung ist die Definition des Begriffs Automatisierungsgrad nicht eindeutig und in verschiedenen Abstufungen zu finden. Manchmal wird er allgemein als *Maß für selbsttätiges Ablaufgeschehen in einem produzierenden Fertigungssystem* bezeichnet, ohne aber weiter auf die Ermittlung dieser Maßzahl einzugehen. Andere Autoren werden etwas genauer, indem sie sagen: *Der Automatisierungsgrad eines Fertigungssystems ist der Quotient aus der Teilmenge der in dem System nach festlegbaren Programm selbstständig ablaufenden Programmschritte bezogen auf die Gesamtmenge der im System ablaufenden Programmschritte*. An anderen Stellen wird diese Definition erweitert indem eine Gewichtung der Programmschritte vorgenommen wird.

$$A^0_{Brödner} = \frac{\sum F_{Aut}}{\sum F_{gesamt}} \qquad A^0_{Hesse} = \frac{\sum F_{Aut} P}{\sum F_{nichtAut} P + \sum F_{Aut} P} \tag{2.2}$$

Auf den ersten Blick scheinen diese Formeln sinnvoll und relativ leicht zu bestimmen. Wenn man nun aber versucht, die Funktionen eines Prozesses herauszuarbeiten und sie zu gewichten, merkt man, dass dies in vielen Situationen nur schwer bzw. mit nicht genügender Genauigkeit möglich ist. Außerdem muss dafür der Mensch als *Funktionenausüber* gesehen werden.

Eine andere Möglichkeit, das Maß der Automatisierung auszudrücken, sind Automatisierungs-

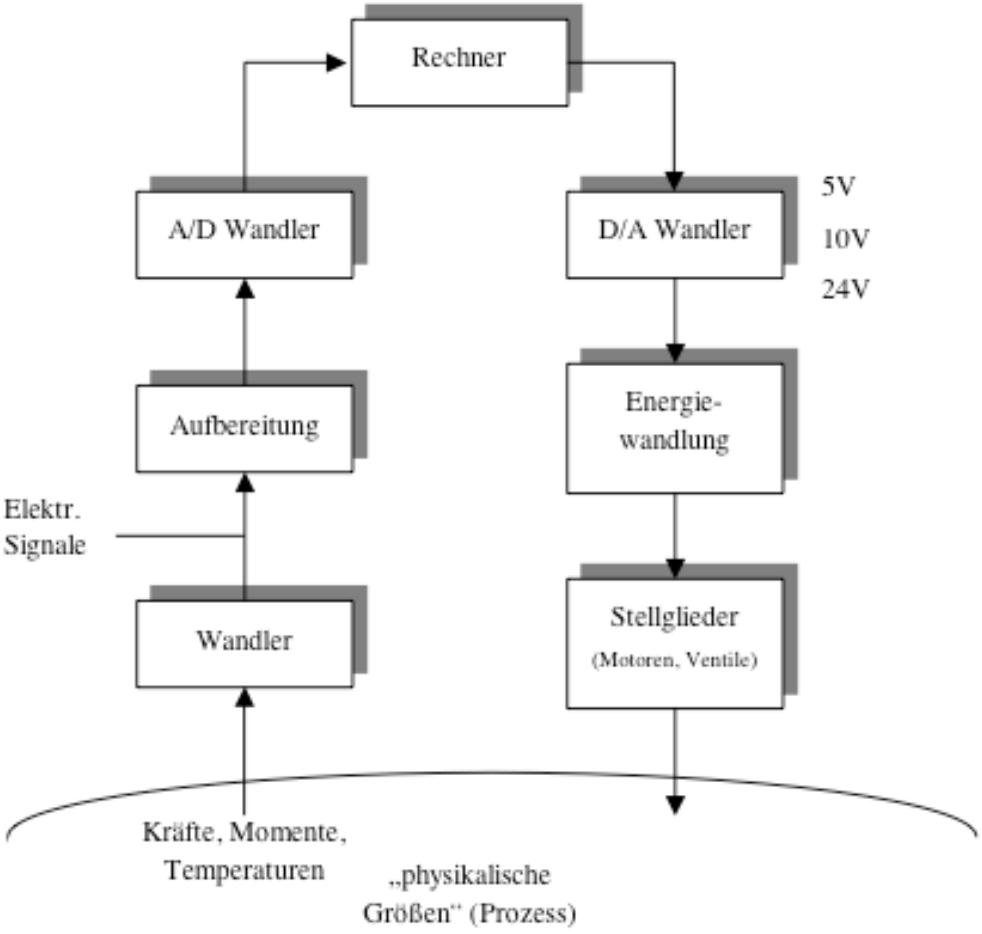


Abbildung 2.8: Prozessrechner

Stufe		Überwachung	Generieren	Auswahl	Implementierung
1	manuelle Kontrolle	Mensch	Mensch	Mensch	Mensch
4	geteilte Kontrolle	Mensch / Computer	Mensch / Computer	Mensch	Mensch / Computer
7	starres System	Mensch / Computer	Computer	Mensch	Computer
10	volle Automation	Computer	Computer	Computer	Computer

Tabelle 2.3: Automatisierungsstufen

	offline	online	open loop	closed loop
indirekt	**X**		**X**	
Erfassung		**X**	**X**	
Steuerung		**X**	**X**	
Regelung		**X**		**X**

Tabelle 2.4: Kopplungsarten

stufen. Dabei erfolgt eine Aufteilung der Aufgaben in die Bereiche Überwachung, Generieren von Wahlmöglichkeiten, Auswahl von Alternativen und die Durchführung der getroffenen Wahl. Diese Aufgaben können dann entweder vom Mensch oder von einer Maschine durchgeführt werden. In Tabelle 2.3 sind beispielhaft einige Stufen aufgeführt:

2.1.10 Rechnereinsatzarten

Neben der Prozesskopplung ohne Rechnereinsatz gibt es verschiedene Möglichkeiten Prozesse zu bearbeiten.

- **off-line-Betrieb**, Betrieb mit indirekter Prozesskopplung, mit dem geringsten Automatisierungsgrad.
- **online- / open-loop-Betrieb**, offen prozessgekoppelter Betrieb, für einen mittleren Automatisierungsgrad.
- **online- / close-loop-Betrieb**, geschlossener prozessgekoppelter Betrieb, mit einem hohen Automatisierungsgrad.

Verschiedene Arten der Kopplung:

Kein Rechnereinsatz

- Die Messgeräte werden vom Bedienungspersonal selbst abgelesen.
- Unter Hinzunahme weiterer Anweisungen von außen (z.B. Vorgaben der Betriebsleitung zum Produktumfang) bedient das Bedienungspersonal entsprechend die Stellorgane.
- Über den Prozessverlauf wird, ebenfalls vom Bedienungspersonal, ein Protokoll erstellt.

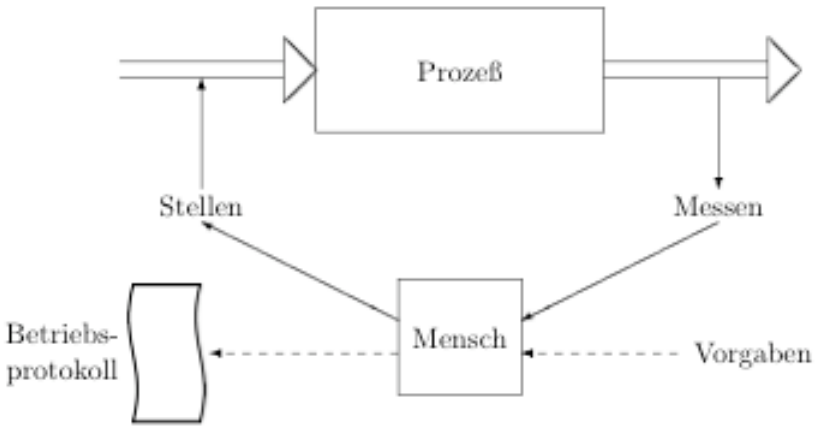


Abbildung 2.9: Kein Rechnereinsatz

- Eine eigene Abteilung ermittelt daraus weitere Anweisungen für den künftigen Prozessverlauf, die dem Bedienungspersonal als schriftliche Unterlagen wieder mitgeteilt werden.
- Ein Rechner wird hierbei nicht eingesetzt.

Indirekte Prozesskopplung (off-line-Betrieb)

- Sowohl das Stellen als auch das Ablesen der Messungen erfolgt hier noch durch das Bedienungspersonal.
- Zusätzlich wird jedoch ein Prozessrechner eingesetzt, der die Auswertungen der abgelesenen und neu hinzugekommenen Informationen vornimmt.
- Die Daten werden vom Bedienungspersonal auf Datenträgern (z.B. Magnetbänder, -platten) abgespeichert und vom Rechner zu einem späteren Zeitpunkt sequentiell verarbeitet.
- Die daraus resultierenden Ergebnisse erhält das Bedienungspersonal wieder in Form von Ausdrucken bzw. Protokollen.
- Rechner und Prozess sind hierbei zeitlich und physikalisch entkoppelt. Typische Aufgaben dieser Kopplungsart sind:
  - statistische Auswertungen von Versuchsreihen
  - Berechnungen der Verluste aus gemessenen Strom- und Spannungsmessungen in einem Elektrizitätsnetz
  - Zuverlässigkeits- und Qualitätsuntersuchungen

Dialogbetrieb (in-line-Betrieb)

- Ein Sichtgerät für das Bedienungspersonal ermöglicht die Dateneingabe und -ausgabe von und zum Rechner.
- Rechner und der Prozess sind zeitlich gekoppelt.

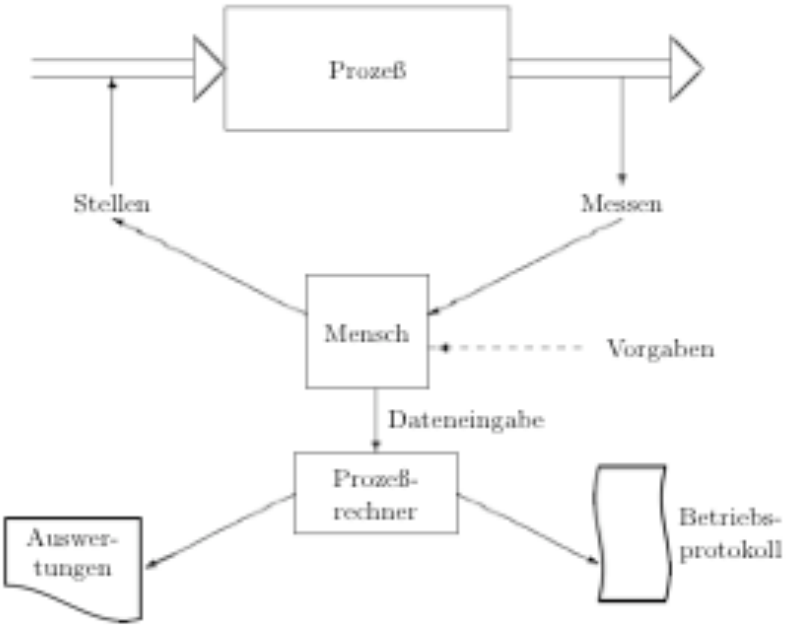


Abbildung 2.10: Indirekte Prozesskopplung (off-line-Betrieb)

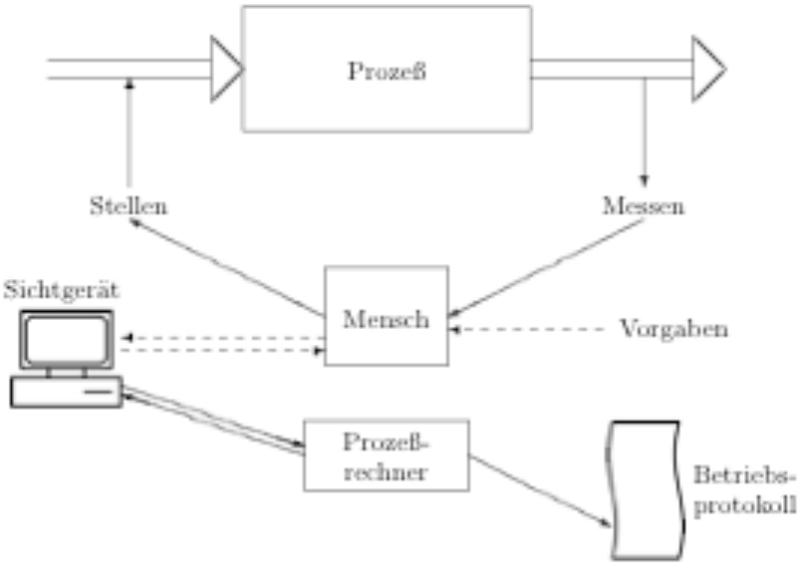


Abbildung 2.11: Dialogbetrieb (in-line-Betrieb)

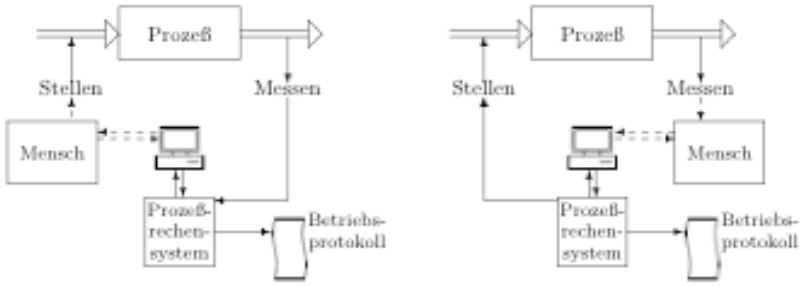


Abbildung 2.12: Indirekte Prozesskopplung (on-line-open-loop-Betrieb)

- Da die Eingriffe in den Prozess hierbei immer noch auf der Seite des Menschen liegen, können Rechnerstörungen in diesem Fall nicht zu gefährlichen Prozesszuständen führen.
- Die Erfahrung des Personals kann in das Ablaufgeschehen mit einbezogen werden.
- Beispiele:
  - Platzbuchungsanlagen,
  - Auskunftssysteme,
  - Fertigungssteuerungen.

Direkte Prozesskopplung (on-line-Betrieb)

- DerRechner wird in einem der beiden Bereiche (Stellen oder Messen) oder in beiden direkt durch Leitungen mit dem Prozess gekoppelt.
- Dies erfordert besondere Anforderungen an die Zuverlässigkeit der Soft- und Hardware des Rechners.
- Das Echtzeit-Betriebssystem muss im Millisekunden- manchmal auch im Mikrosekundenbereich reagieren können.

Indirekte Prozesskopplung (on-line-open-loop-Berieb)

- Nur eine physikalische Verbindung vom Prozess zum Rechner.
- Der Eingriff des Personals erfolgt entweder auf der Seite der Stellgeräte oder aber auf der Seite der Messgeräte.

Indirekte Prozesskopplung (on-line-closed-loop-Betrieb)

- Rechner auf der Seite der Stellgeräte als auch auf der Seite der Messgeräte direkt mit dem Prozess gekoppelt.
- Überwachung des Gesamtsystems erfolgt weiterhin durch den Menschen.
- Es wird jedoch nur im Störungsfall in den Prozessablauf eingegriffen.

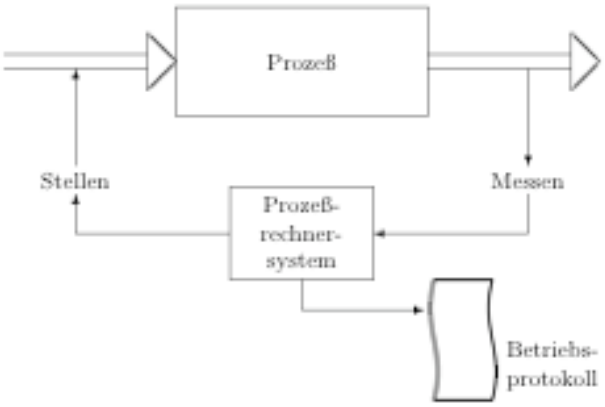


Abbildung 2.13: Indirekte Prozesskopplung (on-line-closed-loop-Betrieb)

2.2 Automatisierung technischer Produkte und technischer Anlagen

2.2.1 Arten von Automatisierungssystemen

Nach [16] wird die Prozessautomatisierung in folgende zwei Gruppen unterteilt:

- Produktautomatisierung: Prozessautomatisierungssysteme, bei denen der technische Prozess in einem Gerät oder einer einzelnen Maschine, meist zur Produktion von hohen Stückzahlen, abläuft.
- Anlagenautomatisierung: Prozessautomatisierungssysteme, bei denen der technische Prozess aus einzelnen Teilvorgängen (Teilprozessen) besteht, die auf größeren, zum Teil auch räumlich ausgedehnten technischen Anlagen ablaufen.

Produktautomatisierung

Prozessautomatisierungssysteme, bei denen der technische Prozess in einem Gerät oder einer einzelnen Maschine abläuft.

Kennzeichnende Kriterien bei der Produktautomatisierung

- Technischer Prozess in einem Gerät oder einer Maschine
- Dedizierte Automatisierungsfunktionen (einfach)
- Automatisierungscomputer in Form von Mikrokontrollern
- Wenige Sensoren und Aktoren
- Automatisierungsgrad 100%, on-line/closed-loop Betrieb
- Sehr große Stückzahlen (Serien- oder Massenprodukte)

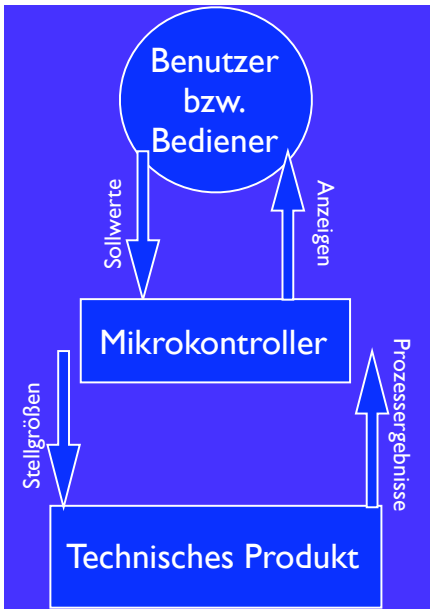


Abbildung 2.14: Produktautomation

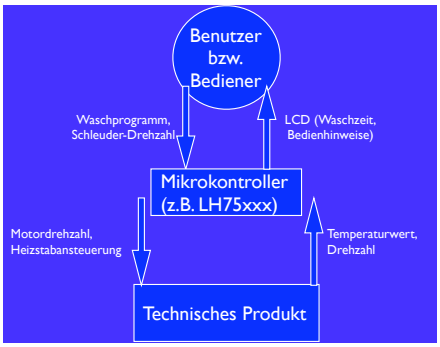


Abbildung 2.15: Waschmaschine

- Engineering- und Softwarekosten spielen eine untergeordnete Rolle, da sie durch die Stückzahl zu dividieren sind.

Anlagenautomatisierung

Prozessautomatisierungssysteme, bei denen der technische Prozess aus einzelnen Teilvorgï¿½ngen (Teilprozessen) besteht, die auf größeren, z.T. auch räumlich ausgedehnten technischen Anlagen ablaufen.

Kennzeichnende Kriterien bei der Anlagenautomatisierung

- Technischer Prozess in einer (oft) räumlich ausgedehnten industriellen Anlage
- Umfangreiche und komplexe Automatisierungsfunktionen
- PC oder Prozessleitsysteme als Automatisierungs-Computersysteme
- Sehr viele Sensoren und Aktoren
- mittlerer bis hoher Automatisierungsgrad
- Einmal Systeme
- Die Engineering- und Softwarekosten sind für die Gesamtkosten entscheidend

Dabei wird das Prozessrechensystem in drei Kommunikationsebenen aufgeteilt:

- Ebene 1 - Feldebene: Prozessnahe Feldbussystem (Kommunikation SPS-Sensoren, -Aktoren),
- Ebene 2 - Prozessebene: Anlagenbus (Kommunikation SPS-SPS, SPS-PC, SPS-Leitrechner),

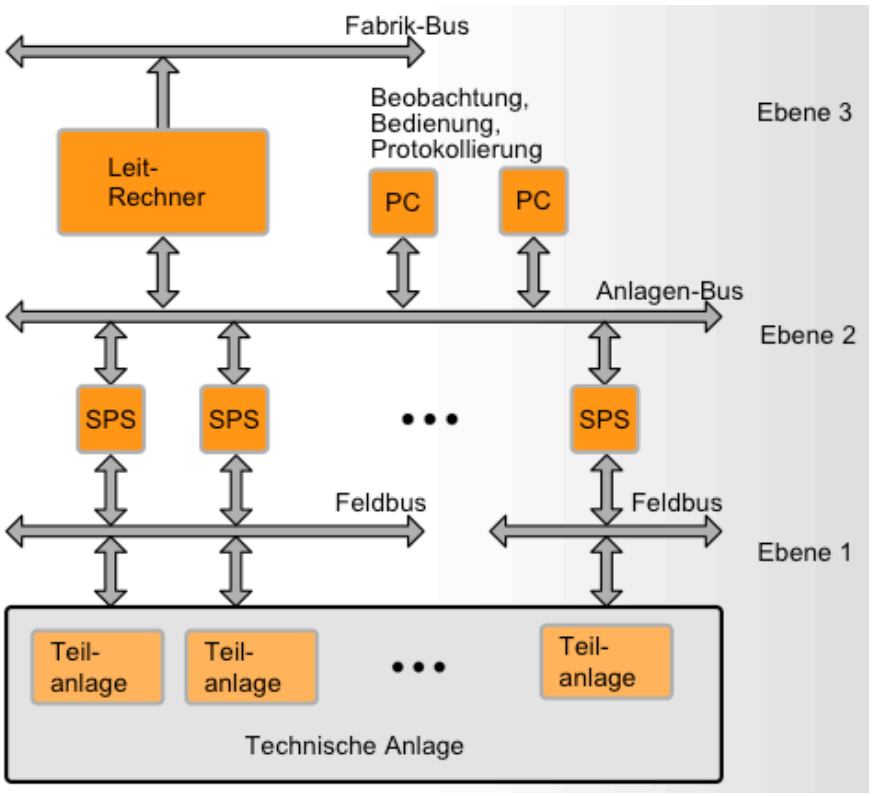


Abbildung 2.16: Anlagenautomatisierung

- Ebene 3 - Betriebsebene: Fabrikbus (Kommunikation Leitrechner-Produktionsplanung).

In diesem Zusammenhang wird auch von Level 1-3 der Automation gesprochen

- Level 1 Automation: Automatisierung auf SPS Ebene (Steuerung, Regelung),
- Level 2 Automation: Prozessvisualisierung (Beobachten, Bedienen, Protokollieren),
- Level 3 Automation: Produktionsplanungssysteme (Statistik, komplexe Modelle).

Beispiele für Produkte bei der Produkautomatisierung	**Beispiele für technische Anlagen bei der Anlagenautomatisierung**
Heizungssystem	Kraftwerksanlagen (Turbinen, ...)
Waschmachinen	Energieversorgungsnetz
Nähmaschinen	Hochregallager
Küchengeräte (z.B. Mikrowelle)	Paketverteilanlage
Fernsehgeräte, Radios	Chemische Reaktoren
Filmkameras	Verfahrenstechnische Anlagen
Alarmanlagen	Stahlerzeugungsanlagen
Spielzeuge	Walzwerkanlagen
Navigationssysteme	Schienenverkehrssysteme
Anrufbeantworter	Straßenverkehrs-Ampelanlagen
Musikinstrumente	Gasversorgungsanlagen
Werkzeugmaschinen	Klär- und Wasserwerke
Messgeräte	Gebäude und Haustechnische Anlagen
Kraftfahrzeuge mit den Teilsystemen Motor, Getriebe, ABS, Abstandswarnsystem, Navigator	Umwelt-Messanlagen

Tabelle 2.5: Beispiele Anlagenautomatisierung

3 Automatisierungsgerätesysteme und -strukturen

3.1 Automatisierungs-Computer

3.1.1 Speicherprogrammierbare Steuerungen

Speicherprogrammierbare Steuerungen (SPS) sind programmgesteuerte, elektronische Automatisierungsgeräte, die für Binär- und Ablaufsteuerungen, Ablaufüberwachung, Datenerfassung und -verarbeitung, Kommunikation und andere eng mit industriellen Abläufen zusammenhängende Aufgaben eingesetzt werden. Sie sind zunehmend auf der untersten Ebene von Automatisierungshierarchien zu finden, d.h. direkt an den zu steuernden Geräten, und werden sehr oft in Architekturen verteilter Prozessleitsysteme integriert.

Die Hardware-Struktur einer speicherprogrammierbaren Steuerung gleicht der Organisation eines Universalrechners. Eine SPS besteht aus einem Prozessor, einem Hauptspeicher, und Ein- / Ausgabeschnittstellen, die durch einen gemeinsamen Bus gekoppelt sind. Trotz dieser strukturellen Ähnlichkeit unterscheidet sich die funktionelle Architektur einer SPS grundlegend von der eines Universalrechners. Die Ein- und Ausgabenschnittstellen ermöglichen es, Informationen zwischen einer SPS und einem technischen Prozess auszutauschen. Ursprünglich wurden SPSen entworfen, um binäre Zustands- und Impulssignale zu verarbeiten. Abhängig vom Typ der SPS kann die Anzahl der Ein- und Ausgaben von 16 in sehr kleinen Steuerungen bis zu mehreren Tausend in großen Systemen reichen. Neuere SPSen bieten die zusätzliche Möglichkeit, analoge Signale effektiv zu verarbeiten.

Normalerweise haben SPSen weder Bedienersichtgeräte noch Massenspeicher, die empfindlich auf rauhe industrielle Umgebungen reagieren. Die Grundbetriebs-Software liegt vollständig im Hauptspeicher. SPSen arbeiten autonom ohne menschliche Eingriffe.

Der Hauptspeicher einer SPS ist in, für verschiedene Zwecke vorgesehene, spezialisierte Segmente unterteilt. Abhängig von der beabsichtigten Anwendung können die Segmente in Bits oder Wörter organisiert sein. Die Grundsegmente enthalten folgende Datentypen:

- **Betriebssystem.** Das Segment ist wortweise organisiert und enthält den Code und die Daten des Betriebssystems; der Systemcode ist normalerweise in einem ROM gespeichert, während die Daten in einem batteriegepufferten RAM-Segment gespeichert sind, das sorgfältig vor Störungen durch andere Programme geschützt ist.

- **Anwendungsprogramme.** Das Segment ist wortweise organisiert und enthält die Anweisungen der Anwendungsprogramme.

- **Anwendungsdaten.** Das Segment ist in drei Bereiche unterteilt, die die von der Maschine mit Messdaten gefüllten Variablen, die Zwischenvariablen und die mit den Stellgliedern assoziierten Ausgabevariablen enthalten. Die Datensegmente der SPSen, die nur logische Funktionen anbieten, sind in Bits organisiert , während bei leistungsfähigeren Maschinen

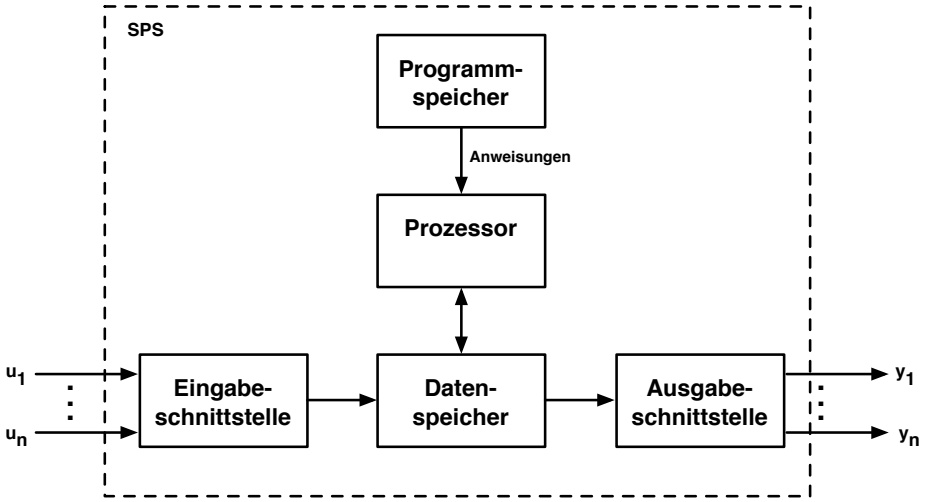


Abbildung 3.1: Architektur SPS

Teile dieser Bereiche wortorientiert sind.

Es gibt eine strenge Entsprechung zwischen den Bitpositionen in den Ein- und Ausgabedatenbereichen und der Anordnung der Ein- und Ausgabesignale an den Anschlüssen und Kabeln, die in eine SPS hinein- und aus ihr herausführen. Alle Ein- und Ausgabeoperationen von SPSen sind implizit. Das bedeutet, dass die Anwendungsprogramme nur auf die Datenbereiche im Speicher zugreifen, während der physikalische Datentransfer ganz von der System-Software oder -Firmware ausgeführt wird. Meistens werden alle Ein- und Ausgabedaten in regelmäßigen Zeitintervallen durch Kanäle mit direktem Speicherzugriff übertragen.

Der Prozessor ist die zur Steuerung einer Maschine und zur Ausführung von Programmanweisungen bestimmte Operationseinheit. Die Anweisungen werden von aufeinander folgenden Worten der Programmsegmente im Hauptspeicher geholt, die vom Programmzähler adressiert werden. Die Daten (Variablen) werden im Datensegment des Hauptspeichers adressiert, wobei eine Vielfalt von Adressierungsmethoden benutzt wird. Die Ausführung der Anweisungen findet im Akkumulatorregister statt. Teilergebnisse werden in einem Stapel gespeichert, der einen internen Speicher des Prozessors bildet. Die für kleine Steuerungen vorgesehenen Prozessoren enthalten nur 1-Bit-Akkumulatoren und können Operationen nur auf Einzelbitoperanden ausführen. Leistungsfähigere Prozessoren haben Akkumulatoren mit 8 . . . 32 Bits und können Operationen sowohl auf Bit- als auch Wortoperanden ausführen.

Grundsätzlich enthält der Befehlssatz eines SPS-Prozessors Anweisungen zum Zugriff auf den Datenspeicher, logische Operationen sowie bedingte und unbedingte Sprünge. Prozessoren für größere Systeme mit digitaler wie auch analoger Signalverarbeitung können zusätzlich numerische Verknüpfungen und Anweisungen zur Kommunikation zwischen Rechnern ausführen.

Der grundlegende Unterschied zwischen der Programmausführung in konventionellen Rechnern und in SPSen ist die Betriebsart: in SPSen bestehen Anwendungsprogramme aus einer Anzahl von Modulen, die in einer ständigen Schleife nacheinander ausgeführt werden. Der Befehlszähler weist der Reihe nach auf alle Wörter des Programmspeichers bis hin zum letzten und beginnt dann wieder mit dem ersten. Anders als in Universalrechnern treten Verzweigungsbefehle nur gelegentlich auf. Die zur Bearbeitung von 1K Befehlswörtern benötigte Zeit wird SPS-Zyklus genannt und ist das wichtigste Leistungsmass speicherprogrammierbarer Steuerungen. Diese zyklische Betriebsart nähert den kontinuierlichen Prozesseingriff klassischer, analoger Steuerungen an. Ein SPS-Zyklus besteht aus zwei Phasen: der Systemphase

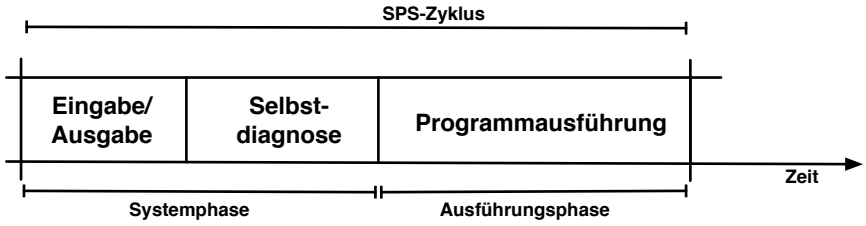


Abbildung 3.2: Zyklus SPS

und der Ausführungsphase. Anwendungsprogramme werden in der zweiten Phase ausgeführt, während die erste Phase implizit und vor dem Anwender verborgen ist. Die Systemphase ist den Ein- und Ausgabeoperationen sowie der Selbstdiagnose der SPS vorbehalten.

Die leistungsfähigsten SPSen erlauben sowohl diskrete Steuerung als auch kontinuierliche Regelung, z.B. nach dem PID-Verfahren. Die Dauer numerischer Verarbeitung kann ziemlich lang sein und bei der Ausführung innerhalb eines SPS-Zyklus' diesen inakzeptabel verlängern. Der Idee des Mehrprogrammbetriebs folgend, sehen solche SPSen daher quasisimultane Ausführungen verschiedener Zyklen vor. Bestimmte Rechenprozesse können zeit- oder ereignisgesteuert aktiviert werden.

Um die Systemleistung zu steigern und die Zuverlässigkeit des Systembetriebs zu erhöhen, wurden Mehrprozessor-SPSen mit dem Potential zu wirklich gleichzeitiger Verarbeitung eingeführt. Es muss betont werden, dass der Bedarf an Parallelverarbeitung nicht aus dem Wunsch nach leistungsfähigerer logischer Verarbeitung entstanden ist, sondern von Anwendungen herrührt, für die SPSen ursprünglich nicht entworfen worden sind. Dazu gehören:

- numerische Verarbeitung (Statistik, technische und kaufmännische Datenverarbeitung)
- Kommunikation (Mensch-Maschine und Maschine-Mensch innerhalb eines Netzes)
- analoge Signalverarbeitung (Messen, Steuern, Regeln)

Mehrprogrammbetrieb erhöht die Komplexität der Systemprogrammierung erheblich. Um die ursprüngliche Vorteile der SPSen, z.B. Einfachheit, Zuverlässigkeit und Wartbarkeit, zu bewahren, wurden für verschiedene Teilaufgaben spezialisierte intelligente Einheiten eingeführt. Grundsätzlich ist jeder Aufruf einer spezialisierten Bearbeitungseinheit implizit und transparent für den SPS-Programmierer. Dieser Ansatz lässt sich an den analogen Ein- / Ausgabeoperationen verdeutlichen: die Funktionen periodisches Abtasten, Sensorüberwachung, Messwertlinearisierung, Skalieren und Schwellwertvergleich können an einen intelligenten Prozessor delegiert werden, der die entgültigen Ergebnisse in einem für diesen Zweck reservierten Speicherbereich der SPS ablegt.

Meistens lassen sich spezialisierte Bearbeitungseinheiten wie folgt klassifizieren:

- numerische Prozessoren für Fest- und Gleitkommaoperationen,
- Überwachungsprozessoren für Busverkehr und Fehlerdiagnose,
- Ein- / Ausgabeprozessoren zur Vorverarbeiten von Eingabesignalen und für Operationen wie Achsenpositionierung, Regelung (z.B. PID), Motorsteuerung usw.,
- Kommuniktionsprozessoren für einfache V.24- oder komplexe Netzkommunikation.

Tatsächlich ist die Mehrzahl der üblicherweise in Universalrechnern zur Prozessautomatisierung verwendeten Hardware-Strukturen in eine Vielzahl von SPS-Konstruktionen übernommen worden.

Auf Grund ihres hohen Spezialisierungsgrades zur Steuerung hauptsächlich diskreter Prozesse wurden SPS für den Gebrauch durch Personen mit geringen Rechner- und Programmierkentnissen entworfen. Hinter der Hardware-Architektur einer Mehrprozessor-SPS ist eine System-Software verborgen, die die Ausführung der Anwenderprogramme koordiniert. Das macht den Gebrauch der Maschine einfach, da die Prozessoren so spezialisiert sind, dass sich ihre Programmierung auf die Auswahl einer Konfiguration und das Setzen von Parametern beschränkt.

Programmierung

Anfangs wurden SPSen vorwiegend in elektrischen Schaltplänen nachempfundenen graphischen Sprachen programmiert. Solche Diagramme stellen eine Abstraktion und eine Formalisierung elektrischer Stromflussdiagramme dar und sind die traditionellen Entwurfs- und Beschreibungsmittel relaisgestützter binärer Steuerungs- und Sicherheitsschaltungen. Ihr fortgesetzter Gebrauch für die SPS-Programmierung erleichtert die Akzeptanz (durch das entsprechende Entwicklungspersonal) der technischen Entwicklung von festverdrahteter hin zu speicherprogrammierter Prozessautomatisierung. SPS-Anwendungsprogramme werden auf externen Geräten entwickelt und übersetzt, die auch den erzeugten Maschinencode für den Gebrauch in SPSen in einige Formen von Festwertspeichern wie (E)EPROMs transferieren können. Eine SPS selbst kann fest eingelagerte Software zum Laden und Ablaufen von Programmen und zum Selbsttesten haben.

Programmiersprachen und rechnergestützte Entwicklungswerkzeuge für SPS-Programme sind zur Zeit im allgemeinen nur in herstellerspezifischer Form erhältlich. Die überlicherweise benutzten Programmiersprachen können in zwei Kategorien unterteilt werden: auf Anweisungslisten basierende Textsprachen, z.B. Assembler-Sprachen ähnelnde Programmiersprachen speziell für einfache Maschinen, und graphische Sprachen in Form von Kontaktplänen oder Funktionsplänen. Wenn andere Operationen als binäre programmiert werden sollen, müssen Kontaktplansprachen bereits funktional erweitert werden.

Um diese Situation zu verbessern, hat die Internationale Elektrotechnische Kommision (IEC) die sehr detaillierte internationale Norm IEC 1131-3 ausgearbeitet, die eine Familie von vier kompatiblen SPS-Programmiersprachen zur Formulierung industrieller Automatisierungsaufgaben definiert. Die Sprachen eignen sich für alle SPS-Leistungsklassen. Weil sie breitgefächerte Möglichkeiten bereitstellen, die umfangreicher sind, als zur Abdeckung des klassischen SPS-Anwendungsgebietes nötig wäre, eignen sie sich auch für den prozessnahen Bereich der Prozessleittechnik. Das gilt jedoch nicht für ihre Kommunikationsmöglichkeiten und die Anwenderschnittstellen.

Die Norm definiert eine Familie systemunabhängiger Sprachen, zwei textuelle und zwei graphische. Die Sprachen sind insofern äquivalent, als sie ineinander überführt werden können:

- **AWL** Anweisungsliste
- **KOP** Kontaktplan
- **FUP / SAP** Funktionsplan / Sequentieller Ablaufplan
- **ST** Strukturierte Text

Es ist das Ziel der Norm IEC 1131-3, die Programmierung in Maschinen-, Assembler- und prozeduralen Sprachen durch die Verwendung objektorientierter Sprachen mit graphischen Anwenderschnittstellen zu ersetzen. Obwohl sie immer noch die niedrigen Sprachen Anweisungsliste und Kontaktplan enthält, betont die Norm deshalb hohe graphische Sprachen für Funktionspläne und Ablaufsteuerung.

3.1.2 Mikrocontroller

Mikrocontroller sind nichts anderes als Prozessrechner, allerdings meist ohne Benutzerperipherie (Tastatur, Bildschirm usw. sind also nicht vorhanden). Die Aufgaben der CPU und die der Prozessperipherie sind auf einem Chip integriert. Dieser enthält also neben Leit- und Rechenwerk und Speicher auch analoge und digitale Ein-und Ausgänge, Interrupteingänge und Zeitgeber.

Durch die heute sehr große Integrationsdichte der Bauelemente sind hohe Rechengeschwindigkeiten möglich. Taktraten liegen bei einigen 100 MHz. Mikrocontroller haben vielfach RISC-Architekturen. Sie haben also nur eine kleine Anzahl von Maschinenbefehlen und arbeiten mit Befehlspipelining. Durch den Einsatz von VLIW-Befehlen (*VLIW: Very Long Instruction Word*) können gleichzeitig mehrere Bearbeitungsstationen im Leitwerk angesprochen werden und damit Werte von CPI < 1 (*CPI: Cycles per Instruction*) erreicht werden. Außerdem sind im Rechenwerk sogenannte MAC-Einheiten (*MAC: Multiply and Accumulate*) vorhanden, die eine sehr schnelle Berechnung von Skalarprodukten gestattet. Die Speicherverwaltung erfolgt entweder in Von-Neumann- oder in Harward-Architektur.

Mikrocontroller wurden speziell für zeitkritische Aufgaben der Mess, Steuerungs- und Regelungstechnik entwickelt. Sie haben alle Merkmale, die ein Einsatz als engebettetes System (Embedded System) erfordert. Insbesondere gestatten Mikrokontroller Reaktionszeiten im Bereich einiger weniger μsec, die von SPSen nicht geleistet werden können, aber z.B. bei der optimalen Steuerung heutiger elektrischer Antriebe, die über Stromrichter gespeist werden, unbedingt erforderlich sind.

3.1.3 Industrie-Rechnersysteme

In den letzten Jahren wurde der PC aufgrund der Erhöhung seiner Solidität mehr und mehr in der industriellen Praxis eingesetzt. Typische Einsatzgebiete für Industrie-PCs sind die Prozessvisualisierung, die Datenhaltung und -auswertung (z.B. Messdatenerfassung, Qualitätssicherung), die Verwendung als Programmiergerät, als Zellenrechner, als Leitsystem, für Verwaltungsaufgaben, als Maschinen- oder Anlagensteuerung oder für die Bildverarbeitung. Für die Nutzung des PC im industriellen Automatisierungsumfeld gibt es mehrere Einschränkungen. Erstens ist die Zuverlässigkeit des PC im 24-Stunden-Dauerbetrieb nicht von Natur aus gegeben. Als Möglichkeit der Erhöhung der Zuverlässigkeit bieten sich unterbrechungsfreie Stromversorgungen (USV), der Ersatz mechanischer Plattenlaufwerke durch Silicon Disks, redundante Laufwerke, Überwachungseinheiten (Watchdog), u.a. an. Zum Zweiten ist die Echtzeitfähigkeit des PC vom Betriebssystem abhängig. Durch den Einsatz von Echtzeit-Betriebssystemen, die zu Windows oder Unix kompatibel sind, kann auch die Echtzeitfähigkeit realisiert werden. Drittens ist die Widerstandsfähigkeit des PC nicht von vorn herein erfüllt. Hier müssen beispielsweise Abdeckungen oder Spezialbildschirme vorgesehen werden. Eine hohe IP-Schutzart kann durch Low-Power-Technik, EMV-gerechte Konstruktion, Kühleinrichtungen, Folientastatur, Vergießen von Leiterplatten oder spezielle Steckkontakte ermöglicht werden. Auch die Einbaufähigkeit des PC in das industrielle Umfeld ist nicht gegeben. Um dies zu realisieren muss eine in der Industrie übliche Aufbautechnik, wie beispielsweise die 19”-Technik verwendet werden. Die

Eignung für die industrielle Kommunikation stellt eine weitere Hürde für den Einsatz des IPC im industriellen Umfeld dar. Dieses Problem kann durch Kommunikationsbaugruppen, wie eine CANbus-Karte oder Industrial Ethernet gelöst werden. Früher war aus genannten Gründen der Preisvorteil des PC als Automatisierungsrechner schnell dahin. Heutzutage ermöglichen jedoch Feldbusse den Anschluss des PC's an das E/A-Werk.

Ein großer Vorteil der Nutzung des PC's als Automatisierungsrechner liegt darin, dass die Bedienung des PC heute schon im Kindesalter angeeignet wird. Aus diesem Grund liegt bei Verwendung des IPC eine gewohnte Bedienoberfläche für den Anwender vor. Außerdem ist die Kompatibilität zum Büro-PC gegeben, was die Verwendung kostengünstiger Soft- und Hardware des Massenmarktes ermöglicht. Ein weiterer Vorteil des PC liegt in seiner universellen Einsatzmöglichkeit durch seine umfangreiche Programmierfähigkeit. Zusätzlich garantiert die Offenheit des IPC, bei dem sowohl Architektur als auch Betriebssystem als Industriestandard etabliert sind, die Herstellerunabhängigkeit. Auch von Vorteil ist die hohe Funktionalität des IPC. Die komfortablen Bedienoberflächen, Massenspeicher, Hochsprachen, Druckeranschluss und die Prozessleittechnik sind einige Beispiele dafür. Ein großer Vorteil des IPC sind die Kostenvorteile gegenüber anderen Rechnerbauformen. Eine Lösung auf IPC-Basis ist oft günstiger als eine mit vergleichbaren Automatisierungsrechnern. Der große Nachteil beim Einsatz des PC in der Automatisierung liegt in den kurzen Innovationszyklen. Die raschen Technologiesprünge widersprechen dem Wunsch des Anwenders nach Kontinuität. An eine Nachliefergarantie von 10-20 Jahren für Erneuerungen und Ersatzteile, die früher durchaus gegeben war, ist heute nicht mehr zu denken. Statt der Komponentengarantie wird heute nur noch eine Funktionsgarantie bis zu 3 Jahren gegeben. Dieses Problem besteht allerdings nicht nur beim IPC, sondern ist heute allgemein in der Elektronik existend. Ein weiterer Nachteil liegt in der E/A-Problematik. Außerdem kann der IPC preislich nicht mit Automatisierungsrechnern des unteren Leistungsbereichs (Klein-SPS, oder Embedded-Controller) konkurrieren.

Für den Einsatz des Industrie-PC in der Maschinensteuerung gibt es zwei strukturell unterschiedliche Möglichkeiten. Einerseits kann der IPC als ergänzende Lösung zum Steuerungsrechner eingesetzt werden. Der PC übernimmt dann anfallende Zusatzaufgaben wie die Speicherung oder Filterung von Daten. Die Steuerung wird weiterhin von einem klassischen Steuerungsrechner übernommen, der bei einer kostengünstigen Lösung mit einer minimalen Anzahl von Komponenten ausgestattet ist. Die Echtzeitfähigkeit und die Ein- und Ausgabemöglichkeit für Prozesssignale des IPC sind hier nicht unbedingt erforderlich. Bei der anderen Möglichkeit übernimmt der IPC alle Aufgaben des Steuerungsrechners und die zusätzlich anfallenden Aufgaben, er ersetzt also den eigentlichen Steuerungsrechner. Bei dieser Lösung ist sowohl die Echtzeitfähigkeit als auch die Ein- und Ausgabemöglichkeit von Prozesssignalen des IPC unbedingt erforderlich.

3.1.4 Single board computer

3.1.5 VMEbus/cPCI/ATCA systems

3.1.6 Board- vs. Box-based Systems

PC-based data acquisition systems are available with a wide variety of interfaces. Ethernet, PCI, USB, PXI, PCI Express, Firewire, Compact Flash and even the venerable GPIB, RS-232/485, and ISA bus are all popular. Which one(s) is/are the most appropriate for a given application may be far from obvious. Perhaps the first question to address when considering a new data aquisition project is whether the application is best served by a plug-in board system (e.g. NI-PCI boards) or an external "box" based system (e.g. Stepper motor controller in a box). This issue has been

a source of much confusion (and competition) over the years, and the decision may be less well defined today than ever. In the early days of PC-based data acquisition, the rule of thumb was: High speed measurements were performed by board solutions, high accuracy was the domain of the external box. Of course, there was a "gray" area in between that could be addressed by either form factor. Today's gray area is much larger than ever before. Board level solutions offering 24-bit resolution are now available as are 6.5 digit DMM boards. On the box side, USB 2.0 is theoretically capable of delivering 30 million 16-bit conversions per second and Gigabit Ethernet will handle more than twice that. Though internal plug-in slot data transfer rates have increased 10 fold in recent years, the typical data acquisition system sample rate has not. Planes and cars don't go much faster now than in 1980 and temperatures and pressures are still relatively slow changing phenomena. Since most application accuracy and sample rates are perfectly within the capabilities of both board and box level solutions, other considerations will determine which solution is best for a given application. Some of these key factors as well as why they are key are listed in the next section.

3.1.7 Tradeoffs and considerations

Distance from the PC to the sensor or measurement

This is a more important consideration than many people realize and it's important for two reasons. First, running long wires from your test system and sensors can be a very expensive proposition, especially in large systems. Running a single communication cable, on the other hand, is inexpensive. Also, each foot of wire connecting your sensor or output to a remote host computer increases your susceptibility to noise. Quiet measurements of 18-bits or greater are almost impossible to obtain using long connection wires. Mounting the DAQ system close to the signal source, however, reduces this noise potential.

Portability

Some systems need to be portable. There are many small, external box devices that meet this need better than trying to drag a desktop or tower PC around. However, don't overlook VME/PXI when portability is a requirement. There are a variety of compact 4- and 6-slot chassis available.

Number of I/O channels

Most people assume the external systems allow for more expandability and may be a better selection for a large system than a plug-in board system. That is often true and most of today's desktop and tower PCs only include a few I/O slots. However, though considerably more expensive than a standard desktop PC, there are a large variety of server and industrial computer chassis providing as many as 16 I/O slots. VME/PXI chassis with up to 18 slots are also available.

PC Obsolescence

External box systems certainly have the edge here. Even if your next PC is functionally identical to your existing computer, do you really want to remove all your I/O boards and install them in your new computer? Also, as technology changes, the slots inside computers change. If your current system has 4 PCI boards in it, are you sure your next PC will have homes for them? Of

course, there is no guarantee your next computer will have the same external connections as your current PC, but the probability is almost certainly higher.

Preferred host computer

It's no secret that laptop computers are becoming ever more popular and their capabilities have expanded to the point where they're not just for road warriors any longer. Your options for developing a plug-in board-based DAQ system around your laptop are pretty limited. There are a variety of PC-Card options available as well as a number of Compact Flash-based devices, but their capabilities and expandability are certainly limited. However, most new laptops come with Ethernet and USB ports, and many include Firewire as well.

Price

The "old" rule of thumb was that all else being equal, a plug-in board based system was likely to carry a smaller price tag. This is no longer the case with some of the lowest cost data acquisition interfaces ever released offering USB or Ethernet interfaces.

Pure speed

The internal buses will almost, by definition, be faster than those based upon an external communications link. After all, if the computer itself can't keep up with the speed of an external communications port, the extra speed is unlikely to be useful. However, only the highest speed applications are beyond the capability of USB, Ethernet, or Firewire.

3.1.8 Prozessleitsysteme

Die Prozessleitebene beschäftigt sich mit der Schnittstelle zwischen dem Automatisierungssystem und dem Bediener der Maschine, bzw. Anlage. Man spricht in diesem Zusammenhang von der Mensch-Maschine-Schnittstelle (MMS) oder auch Human-Machine-Interface (HMI). Im Zusammenhang mit der Bedienung einer Maschine/Anlage spielt das Leiten des Prozesses durch den Prozessführer eine gewichtige Rolle. Auf diese Weise wird die Produktion beeinflusst und zwar in Bezug auf:

- Was wird produziert? (Produktvarianten)
- Wieviel wird produziert? (Menge, Chargengröße)

Um diesen Aufgaben gerecht zu werden existieren die Prozessleitsysteme (PLS). Sie bestehen aus der Gesamtheit vorhandener Automatisierungskomponenten inklusive der wichtigen Mensch-Maschine-Schnittstelle und weiterer Zusatzfunktionen. Zusätzlich zu den allgemeinen Anforderungen an Automatisierungssysteme kommen für Prozessleitsysteme zusätzliche Anforderungen hinzu:

- Redundanz
- Offenheit und Interoperabilität

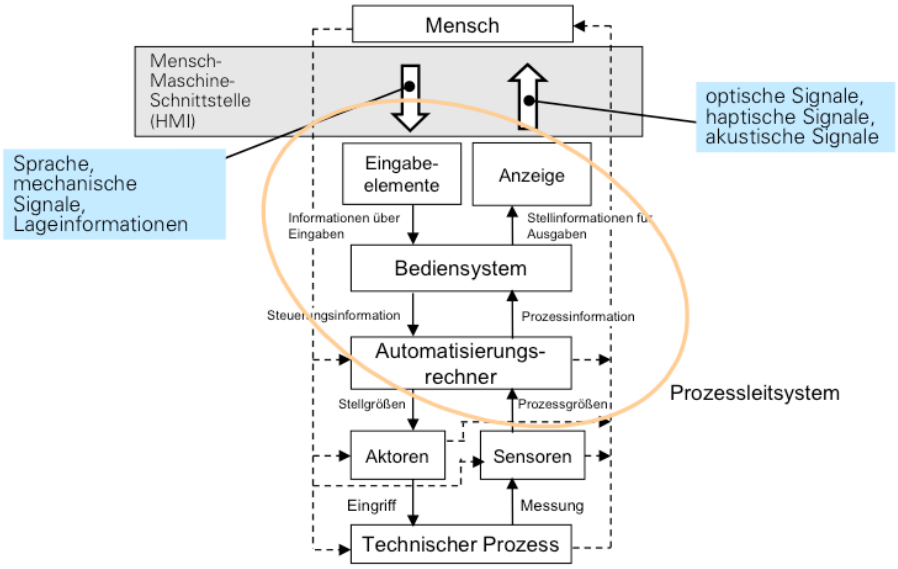


Abbildung 3.3: Prozessleittechnik

- Durchgängigkeit

Ein Prozessleitsystem besteht aus einer Vielzahl von verschiedenen Komponenten auf verschiedenen informationstechischen Ebenen. Daraus leiten sich die Anforderungen an Durchgängigkeit, bzw. Offenheit und Interoperabilität ab. Ein weiterer wichtiger Bestandteil eines Prozessleitsystems ist dabei die Schnittstelle zur Produktionsplanungsebene eines Unternehmens (Manufacturing Execution System, MES). Diese kann z.B. über Gateway-Lösungen realisiert sein. Vor allem spielt jedoch die Semantik der Schnittstelle, d.h. die Art und Weise wie auf Daten zugegriffen werden kann, eine entscheidende Rolle.

Ein weiteres wichtiges Thema im Zusammenhang mit Prozessleitsystemen ist die Redundanz. Im Gegensatz zu Fail-Safe-Prinzip, wo der Prozess im Fehlerfall in einen sicheren Zustand überführt wird, stehen bei redundanten Systemen Ersatzkomponneten bereit, die im Fehlerfall die Aufgabe der fehlerhaften Komponente umgehend übernehmen, so dass Fehlfunktionen keine Auswirkungen auf den Prozess haben. (Das Fail-Safe-Prinzip gilt jedoch grundätzlich immer). Es existieren verschiedene Redundanz-Strategien, die je nach Anforderungen zum Einsatz kommen. Bei einem zweifach redundanten System kann z.B. im Fehlverhalten nur im Sinne von Totalausfall einer Komponnete (z.B. Kabelbruch) festgestellt und daraufhin das Ersatzsystem aktiviert werden. Bei mehrfach redundanten Systemen ist sogar Fehlfunktion im Sinne von falschem Verhalten erkennbar. So wird beispielsweise bei einem 3-fach redundanten System derjenige Messwert verwendet, der von mindestens zwei der drei Komponenten übereinstimmend geliefert wird.

Auszug aus "EINMAL RUPERT UND ZURÜCK"von Douglas Adams:

Klick, summ.
Das große, graue grebulonische Asufklärungsschiff bewegte sich lautlos durch die schwarze Leere. Trotz seiner unglaublichen, atemberaubenden Resisegeschwindigkeit schien es sich vor dem glitzernden Hintergrund aus Milliarden von weit entfernter Sterne nicht zu bewegen. Es war bloß ein einzelner dunkler Fleck, wie erstarrt vor der unendlich feinen Körnung einer leuchtenden Nacht.

An Board des Schiffes war alles, wie seit Jahrtausenden, tiefdunkel und still.
Klick, summ.
Wenigstens fast alles.
Klick, klick, summ.
Klick, summ, klick, summ, klick, summ.
Klick, klick, klick, klick, klick, summ
Hmmm.
Tief im schlummernden Kypernetikhirn des Schiffes weckte ein Überwachungsprogramm der untersten Befehlsebene ein Überwachungsprogramm einer geringfügig höheren Befehlsebene und teilte ihm mit, auf all sein Klicken folge lediglich ein Summen.
Das Überwachungsprogramm der höheren Ebene fragte zurück, was den normalerweise folgen müsse, worauf das Überwachungsprogramm der untersten Ebene antwortete, es erinnere sich nicht, was *genau* folgen sollte, meine jedoch, es müsse eher eine Art entferntes zufriedenes Seufzen sein. Es wisse nicht, was dieses Summen bedeute. Klick, summ, klick, summ. Das sei alles.

Das Überwachungsprogramm der höheren Ebene bedachte dies gründlich und kam zu dem Schluß, daß es ihm nicht gefiehl. Es fragte das Überwachungsprogramm der untersten Ebene, was genau es eigentlich überwache, und das Überwachungsprogramm der untersten Ebene antwortete, auch daran könne e sich leider nicht erinnern, bloß daran, daß es etwas sei, das ungefähr alle zehn Jahre klick, seufz mache, was normalerweise auch einwandfrei funktioniere. Es habe versucht, sein Fehlverzeichnis zu konsultieren, dies jedoch nicht finden können und deshalb das Überwachungsprogramm der höheren Ebene von dem Problem in Kenntnis gesetzt.

Das Überwachungsprogrammm der höheren Ebene beschloß, sein eigenes Fehlerverzeichnis zu konsultieren, um herauszufinden, was das Überwachungsprogram der untersten Ebene eigentlich überwachen sollte.

Es konnte sein Fehlerverzeichnis nicht finden.
Merkwürdig.

Es sah noch einmal nach. Alles was es als Antwort erhielt, war eine Fehlermeldung. Es versuchte, die Fehlermeldung in seinem Fehlermeldungsverzeichnis nachzuschlagen, und konnte auch dies nicht finden. Großzügigließ es einige Nanosekunden verstreichen, um alles noch einmal zu durchdenken. Dann weckte es die Sektorenfunktionsüberwachung.

Die Sektorenfunktionsüberwachung stieß unverzüglich auf Schwierigkeiten und benachrichtigte ihren Überwachungskoordinator, der ebenfalls unverzügllich auf Schwierigkeiten stieß. Komplette Schaltkreise, die teils seit Jahren, teils seit Jahrzehnten geruht hatten, erwachten binnen weniger Millionstelsekunden überall im Schiff flackernd zum Leben. Irgend etwas war irgendwo fürchterlich schiefgegangen, aber keines der überwachungsprogramme konnte sagen. worum es sich handelte. Auf jeder Ebene fehlten lebenswichtige Anweisungen, und auch die Anweisungen für den Fall, daß man feststellte, daß lebenswichtige Anweisungen fehten, fehlten.

Kleine Softwaremodule – Agenten – rauschten angespannt über logische Pfade , gruppierten sich, berieten sich und gruppierten sich neu. Binnen kürzester Zeit wiesen sie nach, daß sämtliche Speichereinheiten des Schiffes bis hinauf zum zentralen Missionsmodul im Eimer waren. Auch mit Hilfe ausgedehntester Speicherabfragen ließ sich nicht herausfinden, was passiert war. Sogar das zentrale Missionsmodul schien beschädigt.

Damit lag die Lösung des Problems auf der Hand. Das zentrale Missionsmodul mußte ausgetauscht werden. Es gab ein weiteres Exemplar, als Backup, ein exaktes Dupikat des Orginals. Der Austausch mußte manuell vorgenommen werden, denn aus Sicherheitsgründen bestand nicht die geringste Verbindung zwischen dem Orginal und seinem Backup. Und sobald das zentrale

Missionsmodul erst einmal ausgetauscht war, würde es selbst die detaillierte Rekonstruktion des restlichen Systems überwachen können, und damit wäre dann alles wieder gut.

Roboter wurden angewiesen, das Backup des zentralen Missionsmoduls aus der massiven Stahlkammer, in der sie es bewachten, zur Installation in die Logikkammer des Schiffes zu bringen.

Dies erforderte einen langwierigen Austausch von Notfallcodes und Protokollen, da die Roboter die Agenten verhören mußten, um sich zu vergewissern, daß deren Instruktionen authentisch waren. Nach langem Hin und Her gaben sich die Roboter aber schließlich zufrieden und betrachteten alle Vorraussetzungen als erfüllt. Sie entfernten die Lagerumhüllung des zentralen Missionsmoduls, trugen es aus der Lagerkammer, fielen aus dem Schiff und trudelten in den leeren Raum.

Das lieferte den entscheidenden Hinweis darauf, was genau nicht stimmte.

Weitere Untersuchungen brachten zügig zutage, was geschehen war. Ein Meteorit hatte ein grosses Loch in das Schiff geschlagen. Was vom Schiff bisher noch nicht registriert worden war, weil der Meteorit haargenau jenen Teil aus der Prozeßsteuerungsvorrichtung des Schiffes herausgeschlagen hatte, der registrieren sollte, ob das Schiff von einem Meteoriten getroffen worden war.

Zuerst mußte versucht werden, das Loch wieder abzudichten. Was sich als unmöglich erwies, weil die Sensoren des Schiffes kein Loch feststellen konnten und die Überwachungsprogramme, die hätten feststellen können, daß die Sensoren nicht richtig funktionierten, selbst nicht richtig funktionierten und darauf beharrten, die Sensoren seien in Ordnung. Das Schiff konnte das Vorhandensein des Lochs lediglich aus dem Umstand ableiten, daß die Roboter eindeutig durch es hinausgefallen waren, wobei sie das Ersatzhirn, mit dem das Schiff das Loch hätte feststellen können, mitgenommen hatten.

Das Schiff versuchte, die Sache vernünftig zu durchdenken, scheiterte und schaltete daraufhin erst einmal alles für einige Zeit auf Null. Daß es alles auf Null geschaltet hatte, bemerkte es natürlich nicht, da es alles auf Null geschaltet hatte. Es war bloß ziemlich überrascht, die Sterne um sich herumhüpfen zu sehen. Danchdem die Sterne zum dritten Mal um es herumgehüpft waren, begriff das Schiff endlich, daß es offenbar alles auf Null geschaltet hatte und daß die Zeit gekommen war, eine schwerwiegende Entscheidung zu treffen. Es entspannte sich.

Dann begriff es, daß es die schwerwiegende Entscheidung noch nicht getroffen hatte und wurde panisch. Erneut schaltete es alles für einige Zeit auf Null. Als es schließlich wieder zu sich kam, schloß es sämtliche Sicherheitsschotten in jenem Bereich, in dem das nicht erkennbare Loch sein mußte.

Es hatte seinen Bestimmungsort ganz eindeutig noch nicht erreicht, dachte es unruhig, aber da es nun nicht einmal mehr den blassesten Schimmer hatte, wo dieser Bestimmungsort überhaupt lag und wie man ihn erreichte, erschien es ihm wenig sinnvoll, die Reise fortzusetzen. Es zog die winzigen Befehlsschnipsel zu Rate, die es aus den Trümmern seines zentralen Missionsmoduls rekonstruieren konnte.

”Deine !!!! !!!! !!!! Jahresmission ist es, !!!! !!!! !!!!, !!!!, !!!! !!!! !!!! !!!!, in sicherer Entfernung !!!! !!!! landen !!!! !!!! !!!! zu überwachen. !!!! !!!!! !!!!. .”

Der Rest war völliger Schrott.

Bevor es entgültig abschaltete, würde das Schiff diese Instruktionen, soweit man sie so nennen konnte, an seine primitiven Unterstützungssysteme weitergeben müssen. Außerdem mußte es

seine Besatzung wiederbeleben.

Damit ergab sich ein weiteres Problem: Während des Tiefschlafs war das Bewußtsein der Besatzungsmitglieder, ihre Erinnerung, ihre Identität nd ihr Wissen um das, weswegen sie unterwegs waren, zur Sicherheit in das zentrale Missionsmodul des Schiffes übertragen worden. Die Besatzungsmitglieder hätten also nicht den blassesten Schimmer, wer sie waren und weshalb sie waren, wo sie waren. Na toll!

Kurz bevor das Schiff dann tatsächlich entgültig abschaltete, bemerkte es noch, daß auch seine Triebwerke im Begriff waren, den Geist aufzugeben.

Das Schiff und seine wiederbelebte, verwirrte Besatzung trieben weiter, kontrolliert von automatischen Unterstützungssystemen, die lediglich interessiert waren, irgendwo zu landen, wo man landen konnte, und zu beobachten, was immer sich zum Beobachten fand. ...

Fehlertoleranz

Es gibt drei grundlegende Typen fehlertolerante Systeme: ausfallsicherheitsgerichtete (fail safe), fehlerkompensierende und fehlermaskierende (fail soft, fail operational) Systeme.

Eine Vorbedingung für ausfallsicherheitsgerichtetes Systemverhalten besteht im Vorhandensein wenigstens eines sicheren Zustandes, der nach der Erkennung eines Fehlers im System erreicht werden kann. Um ein System im Falle eines Fehlers in einen sicheren Zustand zu bringen, müssen alle Systemzustände (nicht nur die normalen Arbeitszustände, sondern auch alle anderen Zwischenzustände) und möglicherweise unsicheren Zustände beobachtet werden. Zur Endeckung von Fehlern werden intern durch Selbstprüfung oder extern durch Verdopplung Formen mehrfacher Redundanz angewandt. Software-Fehler – jedoch nicht notwendigerweise alle – können auch durch Anwendung von Software-Entwurfsdiversität entdeckt werden. Das kann auf verschiedene Arten realisiert werden, z.B. durch serielle Ausführung zweier Programme auf einem einzigen Rechner oder durch parallele Ausführung zweier Programme auf zwei Rechnern. In beiden Fällen entdeckt eine Vergleicherfunktion Diskrepanzen zwischen den beiden Versionen, die einen Fehler im Werte- und/oder Zeitbereich anzeigen. Wenn eine Diskrepanz entdeckt wird, sollte das System durch eine in Hard- oder Software implementierte Notfallprozedur in einen sicheren Zustand gebracht werden. Es gibt zwei Anforderungen an ausfallsicherheitsgerichtete Systeme: ein System darf keine Aktionen ausführen, de es in einen unsicheren Zustand versetzen (passive Bedingung), und das System soll auf Änderungen des gesteuerten Prozesses innerhalb eines definierten Zeitintervalls in sicherer Weise reagieren (aktive Bedingung). Fehlertoleranztechniken, die in ausfallsicherheitsgerichteten Systemen benutzt werden können, sind folgende:

- Zusicherungsprogrammierung (Fehlerüberprüfung)
- 2-Versionenprogrammierung

In einem fehlerkompensierten System gibt es keinen sicheren Zustand oder der Übergang zu einem sicheren Zustand ist nicht möglich oder das System kann keinen sicheren Zustand benutzen, um seine Zuverlässigkeit zu verbessern und seine Funktionalität richtig zu unterstützen. Der Entwurf eines solcen Systems muss für den Umgang mit Fehlern geeignete Maßnahmen, wie z.B. Redundanz, Fehlerendeckungs- und entscheidungsalgorithmen sowie Rekonfigurierbarkeit, bereitstellen, so dass die Systemsicherheit zusammen mit der gesamten oder teilweisen Systemfunktionalität gewährt werden kann. Um das zu erreichen, maskiert oder kompensiert das System bestimmte Fehler. Fehlermaskierung bedeutet Entdecken aufgetretener Fehler und anschließendes Ergreifen von Korrekturmaßnahmen. Wenn Fehler richtig maskiert werden,

haben sie wenig oder keine Auswirkung auf die Funktion eines Systems. Methoden zur Fehlerkompensation und -maskierung können klassifiziert werden, z.B. in Techniken zur Fehlerentdeckung und -korrektur und zur Funktionswiederherstellung sowie in Rekonfigurations- und Redundanzstrategien. Die folgende Liste gibt eine Kategorisierung:

- Fehlerentdeckungs- und Korrekturtechniken
  - defensives Programmieren
  - Fehlerentdeckung und -diagnose
  - Rückkehr zum Handbetrieb
  - fehlerkorrigierende Codes
  - wissensbasierte Überwachung
- Rekonfigurationsstrategien
  - allmähliche Leistungsabsenkung
  - dynamische Rekonfiguration
- Redundanzstrategien
  - aktive Redundanz
  - passive Redundanz (warme Bereitschaft)
  - passie Redundanz (kalte Bereitschaft)
  - N-Versionsprogrammierung
- Wiederherstellungsstrategien
  - Rückkehr zum letzten Wiederaufsetzpunk
  - Sprung zum nächsten Wiederaufsetzpunkt
  - wiederholte Versuche
  - Wiederherstellungsblockschema

Abhängig von dem für eine Software geforderten Integritätsnveau können diese Typen von Fehlertoleranzmethoden zur Herstellung von Hybridsystemen kombiniert werden. Zwei Typen von Hybridsystemen sind wichtig. In einem sicheren Zustand muß eine minimale Funktionalität erhalten bleiben, um Sicherheit zu gewährleisten (teilweise ausfallsicherheisgerichtet). Im anderen Fall besitz ein Hybridsystem einen zeitweilig sicheren Zustand. Unter bestimmten Bedingungen kann es möglich sein, dass ein System weder einen sicheren Zusatnd hat, noch ihn aus bestimmten Gründen erreichen kann.

Zusätzlich zu den in ausfallsicherheitsgerichteten oder fehlerkompensierten Systemen benutzten Methoden gibt es einige andere Strategien, die mit den Unzulänglichkeiten einfacher, die Wirklichkeit darstellender Modelle fertig werden sollen. Strategien hohen Niveaus beinhalten Methoden wie Simulation im Hintergrund (nebenläufige Ausführung), um gefährliche Systemereignisse vorherzusagen und Gegenmaßnahmen im Falle kritischer Abweichung zu ergrei-

fen. Zwei Hauptziele der Strategien hohen Niveaus sind:

- Verhinderung gefährlicher Systemereignisse
- Bereitstellung verhindernder Gegenmaßnahmen im Falle sicherheitskritischer Situationen

Die folgenden Konzepte sollen als zusätzliche Maßnahmen zur Eerfüllung der Anforderungen an sicherheitsgerichtete Systeme in Erwägung gezgen werden:

- Betrachtung der Konsequenzen von Aktionen
- Verhinderung gefährlicher Ereignisse
- Vermeidung gerfähricher Systemzustände
- Vermeidung der Fehlerfortpflanzung
- rechtzeitige Entdeckung gefährlicher Zustände

Für entsprechende Einsatzgebiete existiert die Variante "Klein-PLS". Hierbei handelt es sich um Prozessleitsysteme die nicht zwingendermaßen alle per Definition festgelegten Komponenten enthalten. So kann beispielsweise für manche Anwendungsfälle auf Redundanz verzichtet werden.

Zwischen dem Menschen als Bediener eines Automatisierungssystems, dem System und der vorliegenden Aufgabe gibt es eine Reihe von Wechselbeziehungen. Der Mensch hat bei der Lösung der Aufgabe grundsätzlich konzeptuelle Probleme zu überwinden. Im Zusammenhang mit dem Automatisierungssystem spielt die Bedienbarkeit, die durch Interaktionsprobleme erschwert wird, eine wichtige Rolle. Als Wechselbeziehung zwischen der Aufgabe und dem System kommt der Funktionalität des Systems und der Problematik der Aufgabenimplementierung eine wichtige Bedeutung zu.

Bedien- und Beobachtungskomponente

Wichtiger Bestandteil eines jeden PLS ist die Bedien- und Beobachtungskomponente (BBK). Sie enthält die Schnittstelle zwischen Mensch (in der Bedienerrolle) und der Maschine, bzw. Anlage. Diese Schnittstelle kann auf unterschiedliche Arten ausgeführt sein. Dies zeigt der Wandel der Zeit: vom rein mechanischen Bediensystem früher zum voll computerbasierten System heute. Die Rolle des Menschen im technischen Prozess unterliegt einer permanenten Veränderung. Dennoch bleibt für den Menschen immer eine Aufgabe übrig. Vor der Mechanisierung stellt der Mensch die Energiequelle des Systems dar. Die Bedienung des Systems, beispielsweise einer Handmühle, erfolgt über Kurbeln und Pedale. Im Verlauf der Mechanisierung wird die Energiequelle in die Maschine verlagert. Nach der Mechanisierung hat der Mensch die Aufgabe die Steuerung und Regelung des technischen Systems zu übernehmen. Die Bedienung einer Windmühle zum Beispiel erfolgt über mechanische Stellglieder. Während der Automatisierung werden Steuerungs- und Regelungsfunktionen in die Maschine aufgenommen. Der Mensch dient im Anschluss daran als Überwacher des Systems. Die Bedienung der Automatisierungsanlage erfolgt über Anzeigen und Tasten. Die BBK stellt dem Bediener nicht nur Prozessgrößen zur Verfügung und ermöglicht umgekehrt eine Beeinflussung dieser, sondern beinhaltet weitere Komponenten zur Prozessführung, Auftragsplanung, Archivierung und Rückverfolgung. Um ein Modell eines automatischen Prozesses unter Einbeziehung des Bedieners zu erstellen, muss eine Mensch-Maschine-Schnittstelle (Human-Machine-Interface, HMI) in den Prozessablauf eingefügt werden. Durch diese Schnittstelle wird es dem Bediener ermöglicht mit Hilfe von

Sprache, mechanischen Signalen oder Lageinformationen über die Eingabeelemente des Prozesses auf diesen einzuwirken. Zusätzlich muss es ihm durch Anzeigen des Systems ermöglicht werden optische, haptische oder akustische Informationen über den Prozess zu bekommen. Ein Bediensystem verarbeitet die vom Menschen kommenden Informationen und gibt sie an den Automatisierungsrechner als Steuerungsinformationen weiter. Die primären Aufgaben des Bedieners eines Automatisierungssystems liegen in der Überwachung des Prozesses und dem Erkennen von Fehlfunktionen und dem Eingreifen in den Prozess.

Der Gerätehersteller möchte das Bediensystem so gestalten, dass er es nicht nur an einen Maschinenhersteller verkaufen kann. Seine Anforderungen liegen in der Branchenunabhängigkeit, Design-Optionen und Zulassungen. Der Bediener hat den Anspruch, dass er keine zusätzlichen Handbücher braucht, sondern beispielsweise eine Online-Hilfe in Anspruch nehmen kann. Seine Anforderungen sind Bedienerfreundlichkeit, Störfalldiagnose und Hilfefunktionen. Für den Maschinenhersteller liegen die Anforderungen in Projektierungshilfen, technischen Daten und Schnittstellen. Eine weitere Anforderung ist die Offenheit der Systeme. Hier wird gefordert, dass offene Schnittstellen für Komponenten anderer Hersteller (Ethernet/ Feldbus) vorliegen. Außerdem wird Industrietauglichkeit, so zum Beispiel Resistenz gegen Öle oder Reinigungsmittel gefordert. Der Forderung nach geringem Preis kann durch eine hohe Integrationsdichte oder ein LCD-Display nachgekommen werden. Hohem Bedienkomfort kann durch eine Online-Sprachumschaltung oder durch die Verwendung von Farben und Grafiken entgegengekommen werden. Vernetzbarkeit wird durch Busschnittstellen und Zugriffssicherheit durch die Vergabe von Passwörtern oder durch Schlüsselschalter erreicht. Demzufolge existieren eine Reihe von Normen und Richtlinien, die dafür sorgen, dass den Anforderungen an Bediensysteme Rechnung getragen wird. Diese Standards helfen, die Bedienung zu vereinfachen und Bedienfehler zu reduzieren.

3.2 Automatisierungs-Strukturen

Um die Vor- und Nachteile einer Automatisierungsstruktur bewerten zu können, kann man in Abhängigkeit von der Zahl der Teilprozesse bzw. Automatisierungsfunktionen folgende Kriterien heranziehen:

- die Kosten für die Beschaffung der Geräte, der Verkabelung, der Software, der Pflege und Wartung,
- die Teilverfügbarkeit bei Hardware-Ausfällen oder Software-Fehlern, d.h. die Wahrscheinlichkeit, dass der Betrieb eines Prozesses bis zu einem betrachteten Zeitpunkt nicht durch Ausfälle einzelner Automatisierungseinrichtungen gestört ist,
- die Flexibilität bei Änderungen,
- die Koordinierung der Teilprozesse und Optimierung des Gesamtprozesses sowie,
- die Bedienbarkeit.

Vergleicht man die Kosten für die Beschaffung eines zentralen Prozessautomatisierungssystems mit denen einer dezentralen Struktur, so ergibt sich angenähert das gezeigte Verhalten. Im Gegensatz zu den mit der Zahl der Teilprozesse und/oder der Automatisierungsfunktionen linear ansteigenden Kosten für dedizierte, dezentrale Automatisierungseinheiten sind für einen universellen, zentralen Prozessrechner hohe Anfangskosten aufzuwenden. Die Möglichkeiten zum Anschluss von Prozesssignal- und Prozessbedienperipherie sowie die Fähigkeiten der Kommunikationsgeräte für den Informationsverkehr des zentralen Prozessrechners sind meist

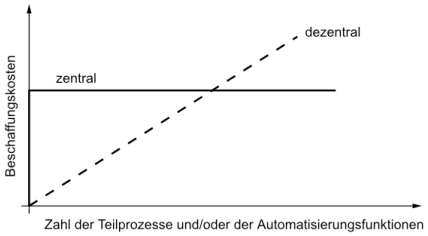


Abbildung 3.4: Vergleich der Beschaffungskosten

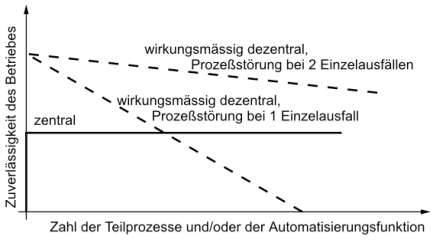


Abbildung 3.5: Vergleich der Zuverlässigkeit

überdimensioniert ausgelegt, so dass weitere Automatisierungsfunktionen ohne oder mit nur geringem zusätzlichen finanziellen Aufwand umgesetzt werden können.

Betrachtet man die Teilverfügbarkeit bei Hardware-Ausfällen oder Software-Fehlern und somit die Zuverlässigkeit des Betriebes so ist bei zentralen Prozessrechnern davon auszugehen, dass alle Störungen auf grund ihrer seriellen Arbeitsweise zu Totalausfällen der Informationsverarbeitung führen können. Da zentrale Prozessrechner schon bei der Beschaffung auf nachträgliche Erweiterungen ihrer Automatisierungsfunktionalität hin ausgelegt sind, kann die Zuverlässigkeit des Betriebes in Abhängigkeit der Anzahl der Automatisierungsfunktionen als konstant angesehen werden. Die Bewertung der Zuverlässigkeit einer dezentralen Struktur ist abhängig von den Auswirkungen, die eine Störung oder der Ausfall einer dezentralen Automatisierungseinheit bezüglich des gesamten technischen Prozesses mit sich bringt. Im Diagramm sind die Kurven zweier Beispiele eingezeichnet. In beiden Fällen nimmt die Zuverlässigkeit des Betriebes mit steigendender Zahl von Teilprozessen ab, da das Gesamtsystem mit entsprechenden Automatisierungseinheiten erweitert werden muss, die wiederum Ursachen für Störungen enthalten können.

Dabei ergibt sich ein stärkerer Abfall der Zuverlässigkeit, wenn durch Ausfall einer Einheit der gesamte Prozess gestört wird, denn ihr Wert berechnet sich als Produkt der Zuverlässigkeitswerte der Einzelgeräte. Im zweiten Fall führt ein gestörtes Einzelgerät noch nicht zu einem völligen Betriebsausfall, weil z.B. die Auswirkungen der Störung gering sind oder Bedienungspersonal vorhanden ist, um die Aufgaben der gestörten Einheit von Hand wahrzunehmen. Erst wenn zwei Einzelgeräte gleichzeitig ausfallen — "gleichzeitig" kann hier auch bedeuten, dass nach dem Ausfall eines Einzelgerätes weitere Automatisierungseinheiten ausfallen, bevor das zuerst ausgefallene instandgesetzt ist —, kommt es zu einem Betriebsausfall. Ein Maß für die Zuverlässigkeit des Betriebes des technischen Prozesses ergibt sind in diesem Fall aus der Wahrscheinlichkeit, dass zwei oder mehr Einzelgeräte gleichzeitig gestört sind. Da diese Wahrscheinlichkeit äußerst gering ist, ist die Zuverlässigkeit des Betriebes für den zweiten Falls sehr hoch.

In den meisten praktischen Anwendung ist die Verkopplung von Teilprozessen nicht so stark, als dass der Ausfall von zwei oder mehr Automatisierungseinheiten einen Totalausfall zur Folge hätte. Daher ist in der Regel die Zuverlässigkeit des Betriebes eines technischen Prozesses mit einer dezentralen Automatisierungsstruktur höher als beim Einsatz eines zentralen Prozessrechners.

Wird dagegen ein zentraler Prozessrechner vorgesehen, der auch die Überwachung, Steuerung und Regelung der Einzelanlagen teilweise oder ganz übernimmt, so sind die Teilprozesse vom Zusammenwirken der Einzelanlagen mit dem Prozessrechner abhängig. Für zentral strukturierte Prozessautomatisierungssysteme gilt damit:

- die Teilaufgaben des Prozesses werden von den gerätetechnischen Anlagenteilen und von

den Programmen des zentralen Prozessrechners bestimmt und

- durch den zentralen Prozessrechner werden bisher autonome Teilprozesse miteinander verknüpft, die dadurch voneinander abhängig werden können.

Durch die letztgenannte Verkopplung von Teilprozessen entstehen neuartige Probleme hinsichtlich folgender Gesichtspunkte:

- **Zuverlässigkeit, Verfügbarkeit und Sicherheit:** Wegen der Zentralisierung und Serialisierung der Aufgabenbearbeitung ist davon auszugehen, dass alle Störungen zu einem Totalausfall der Informationsverarbeitung führen können, der einen Gesamtausfall des Prozesses nach sich zieht.

- **Projektierung und Systementwicklung:** Die bisher angewandten empirischen Verfahren zur Projektierung von Einzelanlagen genügen nicht mehr den Anforderungen, die sich bei der Projektierung komplexer Prozessautomatisierungssysteme stellen. Fehlplanungen und wirtschaftlich unbefriedigende Ergebnisse sind die Folge dieses Mangels.

- **Fehlerdiagnose und Wartung:** Mit der Komplexität der Automatisierungssysteme wachsen die Schwierigkeiten der Fehlerdiagnose und Wartung. Schon die Abgrenzung zwischen Geräte- und Programmierfehlers ist oft nicht leicht.

- **Organisatorische Abwicklung:** Die Verknüpfung autonomer Einzelanlagen zu großen Systemen erzeugt neuartige organisatorische Probleme für die Abwicklung von Prozessautomatisierungsvorhaben.

Bei Aufbau komplexer zentralrechnergeführten Anlagen tritt an die Stelle der Verantwortlichkeit für Einzelanlagen die Verantwortlichkeit fÃ¼r Untersysteme, die aus Geräten und Rechnerprogrammen bestehen. Wegen der engen Verkopplung zwischen den Untersystemen können nun Konfliktsituationen dadurch auftreten, dass eine Beschränkung der Funktionen eines Untersystems im Interesse der optimalen Auslegung des Gesamtsystems erforderlich wird.

Vergleicht man Automatisierungsstrukturen hinsichtlich der Flexibilität bei Änderungen, der Koordination der Teilprozesse und der Optimierung der Gesamtprozesse, so erweisen sich auch hier dezentrale Strukturen als vorteilhaft. Allerdings erfordern dezentrale Strukturen zusätzlichen Aufwand zur Kommunikation der einzelnen Automatisierungseinheiten.

Ein nicht zu unterschätzendes Kriterium zum Vergleich von Automatisierungsstrukturen ist das der Bedienbarkeit und Benutzerfreundlichkeit, denn es steht heute oft im Mittelpunkt des Anwenderinteresses. Dies führt zur Entwicklung entsprechend komfortabler Prozessperipherien und entsprechender Software. Solche Benutzerschnittstellen lassen sich sowohl für zentrale als auch für dezentrale Strukturen verwirklichen. Durch Zuordnung einer dezentralen Automatisierungseinheit zu einem Teilprozess kann jedoch die Transparenz eines komplexen Prozessgeschehens für den Bediener erhöht werden. Die Ursachen von Störungen lassen sich dann sowohl örtlich als auch funktionell besser lokalisieren, abgrenzen und beheben. Dies trägt sehr wesentlich zur Bedienbarkeit und Wartbarkeit komplexer technischer Prozesse in Störfällen bei.

Aus obiger Betrachtung lässt sich als Schlussfolgerung für die geeignete Wahl einer Struktur für die Automatisierung eines technischen Prozesses ziehen:

So dezentral wie möglich, so zentral wie nötig

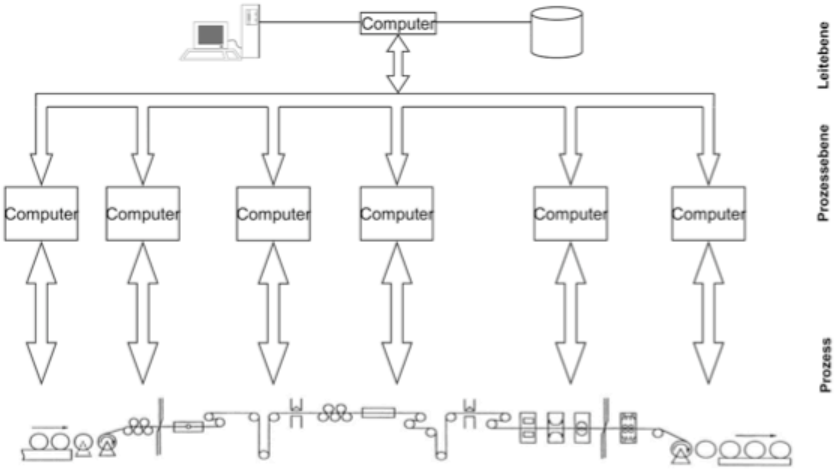


Abbildung 3.6: Kontinuierliche Beizanlage für Metallbänder

In entsprechenden Kombinationen lassen sich damit die jeweiligen Vorteile einer Struktur nutzen, während die jeweiligen Nachteile vermieden werden. Die Wahl dezentraler Strukturen wird sehr stark durch die Entwicklung immer kostengünstiger und leistungsfähigerer Mikroprozessoren begünstigt, die auch auf der untersten Prozessebene eine sehr hohe Flexibilität mit sich bringt.

3.3 Automatisierungs-Hierarchie

Mit den Mitteln der Steuerung und Regelung werden Prozesse automatisiert. Um den Durchsatz zu erhöhen und um Kosten zu sparen, werden verschiedene Hierarchien gebildet, die den Prozess auf unterschiedliche Art und Weise beeinflussen. Folgendes Beispiel gibt eine Einführung, wie auf unterschiedlicher Ebene in den Prozess eingewirkt werden kann.

Als Beispiel für eine Automatisierungs-Hierarchie dient hier eine kontinuierliche Beizanlage für Metallbänder. Die Beizanlage besteht aus verschiedenen Stationen: Metallbänder werden durch Hubbalkenförderer angeliefert und anschließend in eine Abhaspelanlage eingelegt. Danach gelangen die Metallbänder in eine Schopfschere und werden abgerichtet. Die Metallbänder werden in die Beizanlage geleitet und mit einer Abrichtanlage geschnitten und besäumt. Zum Schluß werden die Metallbänder zu einer Rolle aufgehaspelt und über ein Fördersystem verpackt und abtransportiert. Auf der Prozessebene wird beispielsweise in der Abrichtanlage die Einhaltung des Maßes, welches von der Leitebene vorgegeben wurde, gesteuert und geregelt. Eventuelle auftretende Störungen werden weitergegeben an die höhere Hierarchie. So werden auch an der Station, in der die Metallbänder eingelegt werden, auftauchende Fehler an die nächst höhere hierarchische Ebene gemeldet.

Bei der Automatisierung werden die Aufgaben in verschiedene hierarchische Ebenen aufgeteilt:

- Prozessebene und/oder Feldebene
- Operativebene oder System- bzw. Zellebene

- Koordinationsebene oder Prozessleitebene
- Betriebsebene oder Strategieebene

Die Anforderungen an die Realzeitbedingungen nehmen für die einzelnen Ebenen mit Zunahme der Hierarchiestufe ab.

Prozessebene / Feldebene

Die Geräte dieser Ebene wirken direkt auf den Prozess ein oder erfassen Mess- und Statusdaten über den Prozesszustand. Typische Prozessgeräte werden für die Aufgaben wie beispielsweise die digitale oder analoge Ein- und Ausgabe verwendet. So werden zum Beispiel die Daten von den Messgliedern erfasst, mit dem Sollwert abgeglichen und wieder über den Regler zum Einwirken auf den Prozess gebracht. Die Geräte auf dieser Ebene arbeiten meist vollständig parallel und können für gewöhnlich nicht ersetzt werden.

Operativebene

Auf dieser Ebene befindet sich gewöhnlich ein Mikrorechner, der mehrere Gruppen von Reglern, Steuergeräten oder Messgeräten zusammenfasst, deren gemeinsame Aufgabe es ist, die Steuerung einer Fertigungszelle zu koordinieren. Die Operativebene gibt an die Prozessebene Sollwerte aus, empfängt Istwerte und führt Betriebsstatistiken über das zu leitende Prozesssegment. Die ausgewerteten Statistiken gelangen dann an die übergeordnete Koordinationsebene, die dann eine Operativstrategie erarbeitet und diese an die Operativebene übergibt. Die verschiedenen Gruppen der Operativebene können als autonome Systeme arbeiten, und gewährleisten beim Ausfall der Koordinationsebene den Weiterbetrieb der Teilanlagen. Die Reaktionszeit auf dieser Ebene liebt im ms- und s-Bereich.

Aufgaben des Rechners auf dieser Ebene sind das Einlesen und Interpretieren von Istwerten mit anschließsendem Errechnen und Ausgeben von Sollwerten, das Abarbeiten von Regelalgorithmen, periodische Überprüfung der Regler und Messgeräte sowie Melden von Betriebsstörungen an den Operator und übergeordneten Rechner.

Koordinationsebene

Verschiedene Gruppen aus der Operativebene werden auf dieser Ebene zusammengefaßt. Ihre gemeinsame Arbeit wird koordiniert und optimiert. Das Aufgabenspektrum reicht vom Steuern eines Fertigungsbereichs bis hin zum Optimieren der Funktionseinheiten der Operativebene. Die Stellgrößen und Betriebsanweisungen werden an den Operativrechner übertragen. Die Informationen der Operativebene werden für Betriebsstatistiken und zur Optimierung mittels mathematischer oder experimenteller Modelle verwendet. Dies ermöglicht die optimale Vorausplanung und Steuerung der Einheiten auf der Operativebene. Auf dieser Ebene wird im Minuten- und Stundenbereich geplant und gesteuert. Ebenso kann jeder Rechner auf dieser Ebene als autonome Einheit arbeiten und bei kritischen Prozessen seinen Nachbarn auf derselben Ebene nach dem Flying-Master-Prinzip ersetzen.

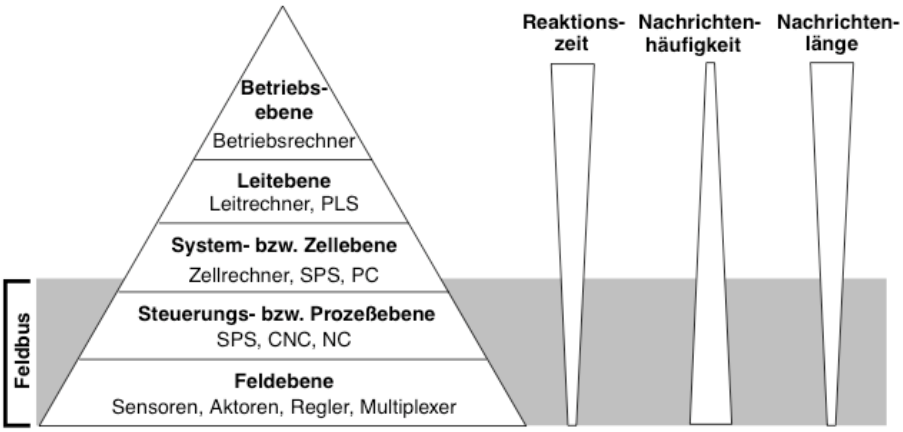


Abbildung 3.7: Kontinuierliche Beizanlage für Metallbänder

Betriebsleitebene

Auf dieser Ebene befindet sich die zentrale Leitung des ganzen Automatisierungssystems. Hier werden sehr komplexe Planungsaufgaben zum ökonomischen und sicheren Betrieb einer Anlage durchgeführt und entsprechende Steuerinformationen an die unteren Ebenen weitergegeben. Für die Planungsaufgaben, die im Stunden oder Tagebereich liegen, sendet die Koordinationsebene Betriebsdaten über die Funktion der einzelnen Untersysteme an die Leitebene. Eine zentrale Aufgabe der Leitebene ist die Erarbeitung von alternativen Automatisierungsstrategien bei sich ändernden Betriebsbedingungen und beim Ausfall eines Rechners der Koordinationsebene oder der Operativebene.

3.4 Verteilte Automatisierungssysteme

Moderne Automatisierungssysteme sind durch eine Verteilung der Automatisierungsfunktionen auf räumliche getrennte Geräte gekennzeichnet. Die Geräteebene mit unmittelbarer Prozesskopplung ist die Feldebene, die zugleich die unterste Kommunikationsebene in der Automatisierungshierarchie darstellt. Auf der Feldebene wird gegenwärtig die noch vorherrschende sternförmige Netzstruktur durch die Busstruktur abgelöst. Der Bus ist ein bidirektionales serielles Übertragungsmedium, das allen Teilnehmern zur Verfügung steht. Ihm wird die Aufgabe gestellt, die mit unterschiedlicher "Intelligenz" ausgestatteten Feldgeräte (Mess- und Stellgeräte, Regler, Bediengeräte, ...) untereinander und mit übergeordneten Leitgeräten (speicherprogrammierbare Steuerungen, Industrie-PCs, ...) zu verbinden. Die Vorteile der Feldbussysteme bestehen hauptsächlich im minimalen Verdrahtungsaufwand und der damit verbundenen erheblichen Kostenreduzierung.

Die Anwendungsbereiche von Feldbussystemen sind die Anlagentechnik, Fertigungstechnik, Verfahrenstechnik, Gebäudeleittechnik und Verkehrstechnik. Die zu steuernden Prozesse sind diskrete Prozesse (Lagerhaltung, Transport) kontinuierliche Prozesse (Raffinerie, Energiegewinnung) oder Kombinationen dieser Prozesse (Druckmaschinen). Die Anforderungsprofile dieser Anwendungsbereiche unterscheiden sich wesentlich und haben zu unterschiedlichen Feldbus-Konzepten geführt. Etablierte Systeme sind z.B. der PROFIBUS (Anlagentechnik, Fertigungstechnik), CAN (Verkehrstechnik) und LON (Gebäudeleittechnik). Hinsichtlich der Kommunikation sind die Automatisierungsgeräte ereignisdiskret arbeitende Subsysteme. Ihr Zugriff

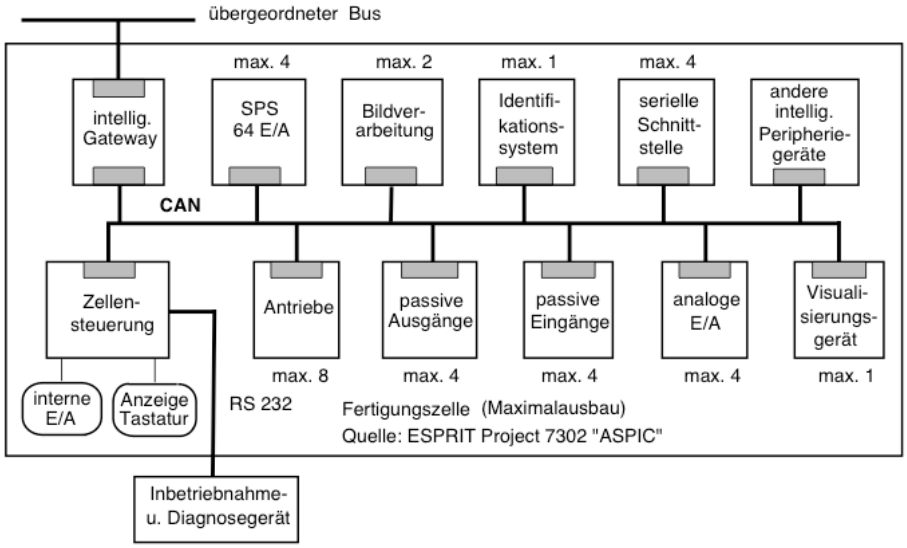


Abbildung 3.8: CAN-Applikation in einer Fertigungszelle

auf den Bus erfolgt nach abgestimmten, für alle Teilnehmer bindenden Regeln, die zusammengefasst als Protokoll bezeichnet werden. Hinsichtlich der unmittelbaren Prozesssteuerung realisieren die Automatisierungsgeräte analoge oder/und digitale Mess- und Stellfunktionen. Eine der wichtigsten Forderungen an die Feldbussysteme betrifft das Zeitverhalten. In der Verkehrstechnik und Anlagentechnik z.B. werden Reaktionszeiten gefordert, die im msec-Bereich und darunter liegen. Von gleichrangiger Bedeutung ist die Sicherheit des Systems bei Störungen oder sprunghaften Lastwechsel. Im Vergleich zu konzentrierten Automatisierungslösungen entstehen zusätzliche Verzögerungen und Beeinflussungen durch das von allen Teilnehmern gemeinsam genutzte Übertragungsmedium.

3.5 Automatisierungsstrukturen mit Redundanz

3.5.1 Hardware-Redundanz

Es gibt mehrere Methoden, um der mangelnden Zuverlässigkeit der Zentralrechnerstruktur zu begegnen:

- Schaffung einer Umschaltmöglichkeit auf redundante Einzelgeräte
- Anwendung redundanter Prozessrechnersysteme (Doppel- oder Mehrrechner) mit Umschaltmöglichkeit im Störungsfall oder
- Einführung dezentraler Prozessrechnersysteme mit weitgehend paralleler Informationsverarbeitung.

Die einfachste Form der Redundanz liegt bereits vor — ohne dass sie allerdings von Betreibern als solche bezeichnet wird — wenn Bedienpersonal parallel zu eingesetzten Rechnern Prozessgrößen überwacht und im Notfall eingreifen kann. Die Aufgaben des Bedienpersonals bezüglich Prozessüberwachung und -sicherung spielen bei Zuverlässigkeits- und Sicherheitsbetrachtun-

gen eine wesentliche Rolle.

Im folgenden soll jedoch nur die Auslegung der Hard- und Software von Automatisierungssystemen betrachtet werden. Hierbei können folgende Formen von Redundanz unterschieden werden.

- redundante Hardware,
- redundante Software,
- Erfassung redundanter Messgrößen, wozu auch auf Grund der Funktion eines technischen Prozesses voneinander abhängige Größen wie z.B. Weg, Zeit und Geschwindigkeit gehören, und
- zeitliche Redundanz, z.B. durch mehrfaches Abfragen des gleichen Messwertes in bestimmten Zeitabständen oder mehrmalige Ausführung des gleichen Programmstückes.

Die beiden letztgenannten Formen der Redundanz sind mit relativ geringem Aufwand zu realisieren und werden häufig in Automatisierungssystemen eingesetzt. Hard- und Software-Redundanz erfordert dagegen einen merklich höheren Aufwand. Letzterer kann jedoch gerechtfertigt sein, wenn dadurch Kosteneinsparungen durch erhöhte Prozessverfügbarkeit erreicht werden. Erhöhter Aufwand muss immer getrieben werden, wenn Ausfälle eine Gefährdung von Menschen mit sich bringen können.

Für die Hardware-Redundanz können verschiedene Einteilungsgesichtspunkte gewählt werden. Bei Betrachtung des Einsatzprinzips unterscheidet man

- **statische oder m-von-n-Redundanz:** die Ergebnisse von n Einheiten, die dieselben Aufgaben bearbeiten, werden miteinander verglichen und Ausgabewerte werden durch m-von-n-Mehrheitsentscheide bestimmt. Die Ergebnisse werden erst dann falsch, wenn mindestens m Einheiten defekt sind, was sehr unwahrscheinlich ist.
- **dynamische Redundanz:** tritt ein Fehler bei einem im Einsatz befindlichen Gerät auf, so wird auf eine Reserveeinheit umgeschaltet.

Betrachtet man die Hardware-Redundanz nach der Arbeitsweise im fehlerfreien Fall, so kommt man zur Unterteilung in

- **blinde Redundanz:** die redundante Einheit ist im fehlerfreien Fall nicht tätig und
- **funktionsbeteiligte Redundanz:** die redundante Einheit führt im fehlerfreien Fall Aufgaben aus.

Methoden zur Steigerung der Zuverlässigkeit von Automatisierungssystemen durch Hardware-Redundanz basieren alle auf der Idee, Reservermodule an den kritischsten Systemstellen vorzusehen. Wenn ein Primärmodul ausfällt oder nicht länger angemessen arbeiten kann, übernimmt ein Reservermodul seine Funktionen. In einem zentralen Prozessrechnersystem ist der Rechner selbst das kritischste Element. Vor diesem Hintergrund wurde die Architektur mit einem redundanten Reserverechner entwickelt.

Dies umfasst zwei Rechner und einen Ausgabeumschalter. Es gibt folgende Möglichkeiten zum Betrieb eines solchen Doppelrechnersystems:

- **Parallelbetrieb:** Beide Rechner lesen die selben Prozessdaten ein und führen dieselben Programme parallel aus. Jedoch sendet nur einer von ihnen Steuerungssignale zum au-

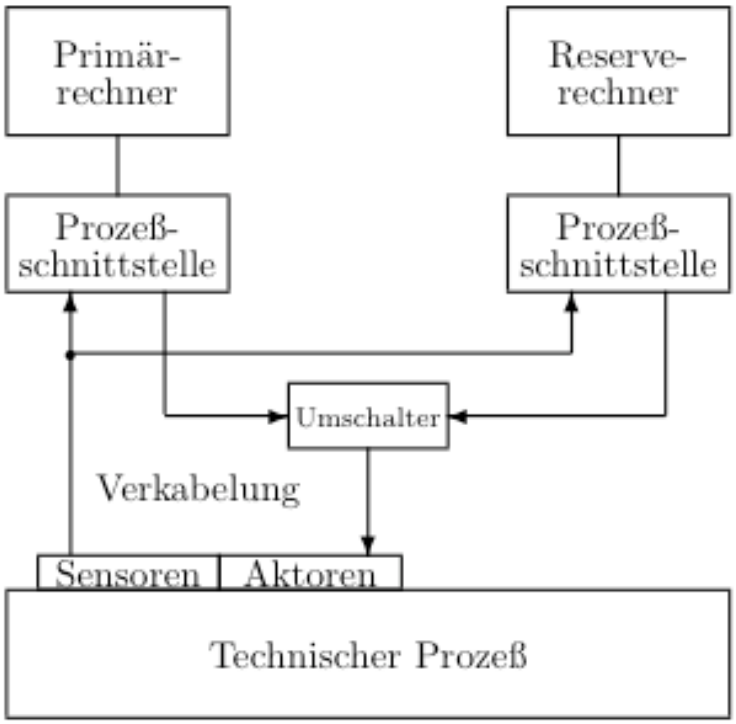


Abbildung 3.9: Zentralisiertes Automatisierungssystem mit einem Reserverechner

tomatisierten Prozess durch den Ausgabeumschalter. Wenn dieser Rechner ausfällt, wird der Umschalter betätigt und der sekundäre Rechner übernimmt. Es bleibt festzuhalten, dass ein Reserverechner den Systemdurchsatz nicht erhöht, da er genau dieselben Verarbeitungen wie der primäre Rechner durchführt.

- **Bereitschaftsbetrieb:** Bei diesem Verfahren werden der Primärrechner als Arbeits- und der Reserverechner als Bereitschaftsrechner bezeichnet. Der Arbeitsrechner nimmt im normalen, ungestörten Betrieb die Automatisierungsaufgaben wahr. Der Bereitschaftsrechner führt Überwachungsrechnungen durch und erhält in kurzen Zeitabständen vom Arbeitsrechner Informationen über die aktuellen Prozesszustände und Zwischenergebnisse. Wird bei der Überwachung ein Fehler erkannt oder meldet sich der Arbeitsrechner nach einer vorgegebenen Zeit nicht mehr, so schaltet der Bereitschaftsrechner den Arbeitsrechner ab und übernimmt dessen Aufgaben an einem festgelegten Einsatzpunkt. Gleichzeitig wird natürlich eine Meldung an das Bedienpersonal gegeben, um die Prüfung und Instandsetzung zu veranlassen.

- **Aufteilung der Automatisierungsaufgaben:** Zur Verringerung der Kosten, die beim Bereitschaftsbetrieb wegen der Auslegung beider Rechner für den gesamten Aufgabenumfang sehr hoch sind, werden bei dynamischer funktionsbeteiligter Redundanz die Automatisierungsaufgaben auf beide Rechner aufgeteilt:

  - der Primärrechner führt diejenigen Automatisierungsaufgaben aus, die zur Aufrechterhaltung der Prozessfunktion unbedingt notwendig sind, z.B. Grenzwertüberwachung, Steuerung, Regelung, während

  - der Reserverechner im normalen, ungestörten Betrieb zwei Arten von Aufgabe wahr-

nimmt:

* er bearbeitet zum einen diejenigen weniger dringlichen Automatisierungsaufgaben, bei deren Ausfall die Funktion des Prozesses zwar beeinträchtigt, aber nicht völlig eingestellt wird, z.B. Optimierungsrechnungen oder Auswertung von Prozessergebnissen, und

* zum anderen überwacht er ständig den ersten Rechner. Wird nun bei der Überwachung ein Fehler im Primärsystem festgestellt, so übernimmt der zweite Rechner voll dessen Aufgaben, wobei er einen Teil seiner eigenen regulären Aufgaben abwirft. Diese Aufgabenübernahme erfolgt allerdings nur einseitig. Bei einem Ausfall des Reserverechners fallen dessen Aufgaben nämlich völlig aus, denn der erste Rechner bearbeitet weiterhin nur seine Aufgaben und übernimmt keine Aufgaben des zweiten Rechners.

Während zum Parallel- und Bereitschaftsbetrieb beide Rechner für die Bearbeitung aller Automatisierungsaufgaben und zur Ein- und Ausgabe aller Prozesssignale ausgelegt sein müssen, genügt bei dynamischer funktionsbeteiligter Redundanz ein Ausbau für die jeweils vorgesehenen Teilaufgaben, wodurch Kosten gespart werden.

Obwohl im Prinzip einfach, ist die Idee der Verdopplung eines Automatisierungssystems außergewöhnlich schwer zu implementieren und die Kosten der Implementierung sind sehr hoch. Ein genauerer Blick zeigt, dass es eine Anzahl möglicher Störungsquellen im System gibt. Diese sind

- Sensoren und Aktoren
- Verkabelung (gewöhnlich fällt ein Kabel nicht aus, aber es kann aufgerissen oder durchgeschnitten werden — versehentlich oder böswillig),
- Umschalter,
- Prozessschnittstelle,
- Zentralrechner und
- zusätzliche Elemente wie Stromversorgung oder Kühlung (sofern benötigt).

Alle diese Elemente müssen ordnungsgemäß funktionieren, um einen korrekten Betrieb des Automatisierungssystems zu gewährleisten, und jedes defekte Gerät kann einen Prozessstillstand verursachen. Um die Systemzuverlässigkeit real zu erhöhen, müssen alle Elemente, oder zumindestens alle sicherheitskritischen, verdoppelt werden. Jedoch können weder der Umschalter noch die Aktoren verdoppelt werden. Es ist zum Beispiel nicht möglich, ein Ventil durch Montage eines zweiten am selben Rohr abzusichern: werden sie eines hinter dem anderen am selben Rohr montiert, dann schließt bereits eines von ihnen das Rohr; werden sie anderenfalls parallel an zwei Zweigen des Rohres montiert, dann öffnet bereits eines von ihnen permanent den Fluss durch das Rohr.

Die Anwesenheit eines Ausgabeumschalters schafft das Problem der Steuerung seiner Position. Es ist offensichtlich, dass der Schalter nicht vom Primärrechner gesteuert werden kann, da bei seinem Ausfall umgeschaltet werden soll. Aber er kann auch nicht vom Reserverechner gesteuert werden, denn dann könnte er als Ergebnis einer Fehlfunktion dieses Rechners umgeschaltet werden. Es sollte auch, wie bereits bemerkt, berücksichtigt werden, dass der Schalter selbst versagen kann.

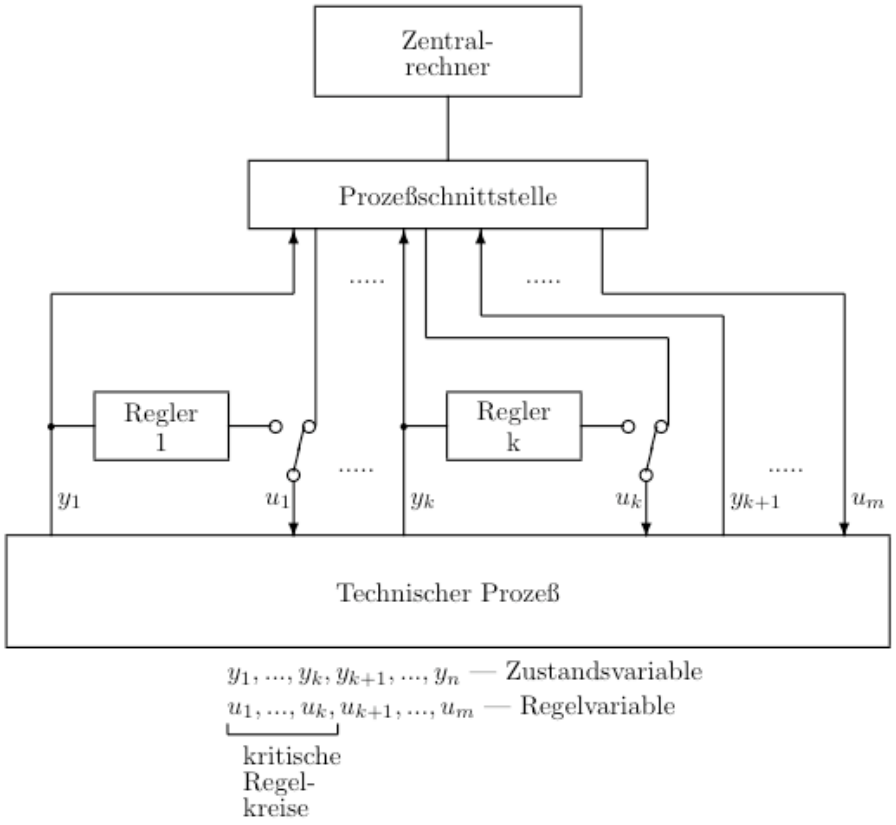


Abbildung 3.10: Automatisierungsstruktur mit Einzelgerätereserve

In der industriellen Praxis wird die Konfiguration mit einem Reserverechner nicht als kosteneffektiv angesehen und deshalb selten benutzt. In den seltenen Fällen des Einsatzes eines Reserverechners kann das Umschaltproblem abhängig von den Anwendungsmerkmalen in einer von zwei Weisen gelöst werden. Eine dieser Methoden ist dann anwendbar, wenn die erforderliche Geschwindigkeit der Rechneraktion relativ gering ist, so dass der Umschalter manuell von einem menschlichen Bedieners betätigt werden kann, von dem erwartet wird, dass er eine Rechnerfehlfunktion erkennt. Anderndfalls wird eine spezielle Umschaltsteuerung verwendet, die beide Rechner überwacht — z.B. indem sie von beiden in regelmäßigen Zeitabständen Bereitschaftsmeldungen empfängt. Wenn eine Bereitschaftsmeldung nicht rechtzeitig kommt, dann wird der entsprechende Rechner als ausgefallen angesehen. Man kann sich klarmachen, dass diese Methode relativ sicher ist, da eine Fehlfunktion der Umschaltsteuerung nicht gefährlich ist, sofern weder der Primär- noch der Reserverechner ausgefallen sind. Soweit es der Umschalter betrifft, wird angenommen, dass seine Zuverlässigkeit höher als die der Rechner ist.

Ein alternativer Ansatz ist nützlich, wenn ein System logisch in eine Anzahl von Regelkreisen aufgespaltet werden kann. Er besteht darin, nur einzelne Kreise anstelle des ganzen Automatisierungssystems zu verdoppeln. Der Ansatz ist sehr flexibel, da in der Regel nicht alle Teilaufgaben des Systems von gleicher Relevanz sind. Gewöhnlich können nämlich einige von ihnen vorübergehend zurückgestellt werden, wenn ein Ausfall auftritt. Grundsätzlich kann diese Methode auf kontinuierliche Regelkreise und zur binären Steuerung angewandt werden. Jedoch wird sie meistens für kontinuierliche Regelkreise benutzt, für die PID-Regler einsetzbar sind. Gelegentlich werden nicht nur die Regler, sondern auch die Sensoren und Kabel verdoppelt. Ein einzelner Kreis kann dann manuell von einem Bediener oder automatisch von einer speziellen Steuerung umgeschaltet werden.

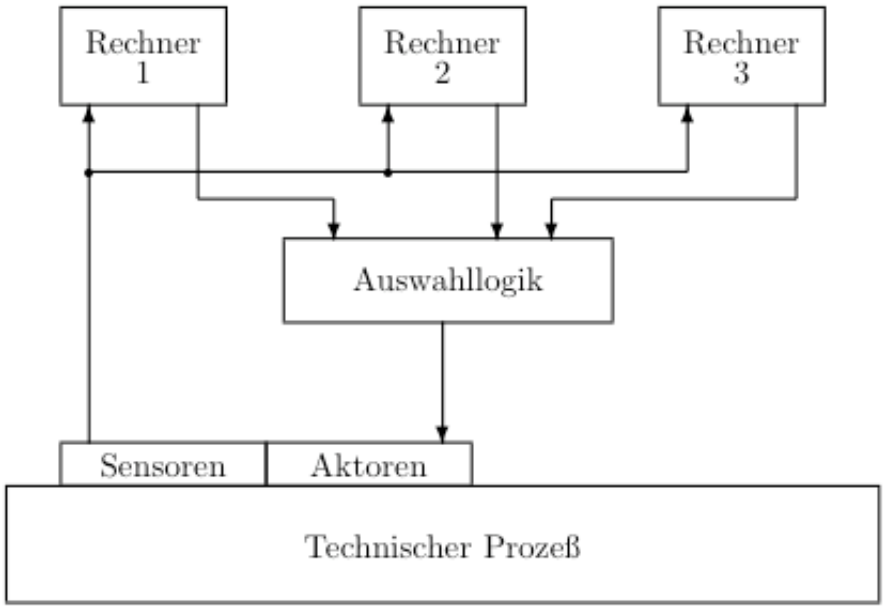


Abbildung 3.11: Parallele Rechner mit Ergebnisauswahl

Die Idee, *Einzelgeräte als Reserve* vorzuhalten, ist nicht auf zentralisierte Rechnersysteme beschränkt, sondern kann ebensogut in jeder anderen Systemarchitektur angewandt werden, z.B. der Einzelgerätetechnik selbst. Sie hat sich als kosteneffektiv erwiesen und wird in der industriellen Praxis häufig angewandt.

Eine mit Doppelrechnersystemen angestrebte hohe Zuverlässigkeit des Betriebes technischer Prozesse genügt nicht, wenn bei Ausfällen Gefahren für menschen eintreten können. Zusätzlich wird in diesen Fällen die Forderung gestellt, dass gefährliche Einzelfehler nicht auftreten dürfen. Als gefährlich werden dabei solche Fehler bezeichnet, die die Ausgabe falscher Prozesssignale verursachen oder eine Sicherheitsoperationen wie eine Abschaltung oder Störungsmeldung verhindern könnten. Ähnlich gilt für Raumfahrt- und Militäranwendungen, z.B. zur Luftverteidigung oder Steuerung balistischer Raketen, wo höchste Zuverlässigkeit verlangt und Kosten als weniger wichtiger Faktor betrachtet werden.

Die geforderte Zuverlässigkeit und Sicherheit kann durch den Einsatz von drei oder mehr unabhängigen Rechnern erreicht werden, die parallel arbeiten. Alle Rechner erhalten dieselben Daten von einer gemeinsamen Umgebung und müssen auf dieselben Anforderungen antworten. Ihre Ergebnisse werden verglichen und die, die von der Mehrheit der Rechner geliefert wurden, werden von einer Auswahllogik ausgewählt (*m-von-n-Redundanz*). Die Wahllogik kann zwar per Programm durchgeführt werden, wird aber als kritischer Teil der Systems üblicherweise durch spezielle ausfallsicherheitsgerichtete Techniken implementiert, die bei Fehlfunktionen auf jeden Fall einen sicheren Zustand herbeiführen.

Es ist wert festzuhalten, dass die Rechner in einem Auswahlsystem nicht vom selben Typ zu sein brauchen. Sie müssen nur dieselbe Verhaltensfunktionalität implementieren, während die Einzelheiten der Hardware wie auch die Software-Implementierung verschieden sein können. Dies erhöht die Immunität des Systems gegen Entwurfs- und Codierungsfehler.

Parallele Systeme mit Auswahl sind selten in industriellen Prozessautomatisierungsanwendungen, insbesondere in der kontinuierlichen Regelung, zu finden. Jedoch können sie in natürlicher

Weise in Umgebungen mit diskreten Ereignissen und in Kommandosystemen eingesetzt werden.

Bei Einsatz redundanter Prozessrechensysteme muss immer gewährleistet sind, dass nach Ausfall einer Einheit

- der Ausfall von den anderen Einheiten erkannt wird,
- Daten gesichert werden,
- die Automatisierungsfunktionen durch andere Einheiten übernommen werden,
- eine Meldung an das Betriebspersonal ausgegeben wird.

Bis jetzt hat sich die Diskussion in diesem Abschnitt auf Methoden zur Gewährleistung kontinuierlicher Betriebsbereitschaft von Prozessautomatisierungssystemen beim Auftreten von Ausfällen konzentriert. Implementierungen dieses Ansatzes der Fehlertoleranz haben sich als schwierig und teuer herausgestellt. Der zweite Ansatz ist der, einen automatisierten Prozess in einem sicheren Zustand und zu minimalen Kosten anzuhalten.

3.5.2 Software-Diversität

Obwohl sicherheitsanalytische Techniken notwendig und sinnvoll sind, kann Software-Sicherheit nicht allein durch Analysen und Verifikationen gewährleistet werden: die Analysetechniken sind so komplex, dass sie selbst fehleranfällig und ihre Kosten möglicherweise untragbar sind und dass eine hochzuverlässige Eliminierung aller Gefahren starke Leistungseinbußen zur Folge haben könnte. Daher müssen Gefahren auch während der Benutzung von Software kontrolliert werden.

Programmfehler sind grundsätzlich systematischer Art. Um diesen systematischen Programmfehlern begegnen zu können, ist daher der Einsatz diversitärer Software prinzipiell kein sinnvolles Konzept: Software sollte eigentlich korrekt sein. Auf Grund ihrer hohen Komplexität und der Nichtverfügbarkeit geeigneter Methoden für Korrektheitsnachweise gibt es jedoch oft keinen anderen Weg, als systematische Fehler durch diversitär gestaltete Software erst zur Laufzeit zu entdecken und zu entfernen. Diversitäre oder "mehrkanalige" Software muss nicht durch mehrkanalige Hardware unterstützt werden. Diversität ist wegen der viel niedrigeren Zuverlässigkeit von Software im Vergleich zur Hardware oft auch ein nützliches Konzept in Einkanalsystemen. Dann wird natürlich die Systemleistung durch die Ausführungszeitanforderungen der mehrfachen "Software-Kanäle"herabgesetzt.

Eine auf das Minimum reduzierte Form der zweikanaligen Architektur mit internem Ergebnisvergleich stellt die Kontroll-/Schutzsystemstruktur dar. Das Prinzip der Überwachungsfunktion des Schutzsystems können angenommene statische Grenzen, Simulation des Systemverhaltens mit Ergebnisvergleich oder Identifikation der aktuellen Parameter im Vergleich zu den erwarteten darstellen. Daher ist die Kontroll- / Schutzsystemstruktur auch eine Form diversitär redundanter Systemimplementation. Sie kann durch die deutliche Trennung von Funktions- und Sicherheitsaufgaben Entwicklungs- und Anschaffungskosten reduzieren.

Die am häufigsten zur sicheren Prozessdatenverarbeitung eingesetzte reguläre diversitäre Architektur ist die zweikanalige. Ein zweikanaliges Software-System ist characterisiert durch:

- zwei diversitäre fehlerarme Software-Einheiten, die dieselbe Funktion realisieren,

- Vergleich der Ausgabewerte und Zwischenergebnisse beider Software-Einheiten, um Restfehler zu erkennen und deren Auswirkungen sicherheitsgerecht zu beeinflussen.

Auf zwei verschiedenen Rechnern laufen demnach zwei verschiedene, fehlerarme Programme gleicher Funktion. Ihre Ausgaben und Zwischenresultate werden zur Entdeckung von noch vorhandenen Fehler, von einem in einer ausfallsicherheitsgerichteten Technik implementierten Vergleicher überprüft. Bei der Endeckung von Diskrepanzen schaltet der Vergleicher die Rechner ab und überführt den Prozess in einen sicheren Zustand. Es muss beachtet werden, dass echt ausfallsicherheitsgerichtetes Verhalten nur in Hinblick auf Hardware-Fehler erzielt werden kann. Wenn ein Prozess keinen sicheren Rückfallzustand (im Allgemeinen die Abschaltung) hat, muss beim Systementwurf immer ein Hardware-Kanal mehr als für Prozesse mit sicheren Zuständen vorgesehen werden.

Die in beiden Kanälen eines zweikanaligen Systems laufende Software muss fehlerarm sein. Die für einkanalige Lösungen verlangte Fehlerfreiheit braucht hier jedoch nicht erfüllt zu sein. Das bedeutet beträchtliche Kosteneinsparungen, sollte aber nicht zu geringeren Bemühungen bei der Qualitätssicherung führen. Der zusätzliche Aufwand einer zweikanaligen Lösung liegt in der Realisierung zweier diversitärer Software-Einheiten. Die Diversität beider Programmeinheiten kann, da Software-Diversitär entweder durchgängig eingesetzt oder nur auf wichtige, spezifische Module konzentriert wird, folgende Formen haben:

- vollständige Diversität,
- gezielte Diveristät.

Vollständige Diversität

Vollständige Diversität ist durch die Verwendung aller möglichen Diversitätsarten charakterisiert. Die dabei eingesetzten Typen von Diversität sind auf das Erkennen der möglichen verbliebenen Fehler hin ausgerichtet, die als solche identifiziert worden sind. Im allgemeinen verlässt man sich bei der Entwicklung diversitärer Software nicht auf zufällige erzielte Vielfalt, sondern versucht, Vielfalt zu erzwingen. Genauer gesagt können Unterschiede im Hinblick auf folgende Aspekte erzielt werden:

- Entwurfs- und Entwicklungsgruppen an verschiedenen Orten ohne Kontakt untereinander mit unterschiedlichen Testdatenmengen,
- unterschiedliche Spezifikationsmethoden und -sprachen,
- unterschiedliche Beschreibungen derselben Spezifikation,
- unterschiedlicher Entwurf und unterschiedliche Entwurfsbeschreibungen,
- Gebrauch verschiedener Entwicklungswerkzeuge, Programmierumgebungen, Programmiersprachen und Übersetzer,
- Gebrauch unterschiedlicher Rechnertypen,
- unterschiedliche Laufzeit- und Betriebssysteme sowie andere Standard-Software und Software-Hilfsmittel,
- Optimierungskriterien: Laufzeit- oder Speicheranforderungen,
- unterschiedliche Implementierungen:

  - unterschiedliche Algorithmen und Methoden,
  - unterschiedliche Datenstrukturen, -formate und Zugriffsmethoden,
  - unterschiedliche Eingabedaten,
  - Auswertung einer Funktion oder ihrer Inversen,
  - unterschiedlicher Ablauf- und Zeitverhalten,
  - unterbrechungsaktivierte Ablaufsteuerung oder zyklische Ereignisabfrage,
- unterschiedliche Verifikationsmethoden.

Zur Implementierung diversitärer Programmeinheiten sind folgende Methoden aus der Literatur bekannt:

- Gebrauch diversitärer Spezifikationen und diversitärer Prozessmodelle, z.B. Betriebs- oder Sicherheitsmodell,
- N-Versionenprogrammierung, d.h. vollständig verfügbare Diversität,
- Rücksetzpunktschema, d.h. aufrufbare Diversität,
- Zeitdiversität, d.h. Wiederholung einer Funktion zu verschiedenen Zeitpunkten zur Erkennung transienter Fehler,
- Methode der "ungenauen Ergebnisse", d.h. Diversität verschiedener Qualität zu unterschiedlichen Kosten,
- Vergleich der Realität mit einem Modell, z.B. mit einem Prozessbeobachter, als diversitäre Sicherheitsmaßnahme.

Gezielte Diversität

Gezielte Diversität wird durch die Anwendung bestimmter Diversitätsarten erreicht. Diese sind auf die Erkennung von Restfehlern hin ausgerichtet. Restfehler sind durch die verwendeten Verfahren der Software-Entwicklung und -Tests bestimmt. Wird z.B. ein rechnergestütztes Spezifikations- und Entwurfsystem benutzt, das nur Software-Strukturen nach dem Verfahren der strukturierten Programmierung zulässt, so sind Fehler, die mit anderen Strukturen zusammenhängen, ausgeschlossen. Die Einschränkung der Benutzung höherer Programmiersprachen auf nur ganz bestimmte Typen und Formen von Anweisungen kann hinsichtlich der Restfehler einen großen Vorteil bedeuten. Solche Einschränkungen müssen aber konsequent befolgt werden. Die Einhaltung der Einschränkungen kann mit einem Rechner leicht überprüft werden. Beispiele solcher Einschränkungen sind:

- keine Sprunganweisungen,
- keine Programmunterbrechungen in Unterprogrammen,
- keine zwei- und mehrdimensionale Felder,
- keine Auswahlanweisungen,

- keine Indexberechnungen durch arithmetische Ausdrücke.

Ist die Kategorie der Restfehler bekannt, d.h. sind die möglichen Restfehler eingekreist, so ist festzulegen, welche Diversitätsarten erforderlich sind, um diese Restfehler zu erkennen. Das ist kein leichtes Problem, weil der Zusammenhang zwischen verwendeter Diversitätsart und der dadurch erkennbaren Fehler sehr komplex ist. Da hier kaum Erfahrungen vorliegen, neigt man sehr schnell dazu, die Kategorie der Restfehler nicht einzuschränken, d.h. alle Fehlerarten als nicht ausschließbar zu betrachten, und daher möglichst viele Diversitätsarten vorzusehen, Diese Strategie ist sicherlich falsch, da man einerseits sehr viel Aufwand für Ausschluss und Behebung von Fehlern während der Entwicklung aufbringt und man andererseits einen ebenso großen Aufwand treibt, um alle möglichen Fehler während des Betriebes rechtzeitig zu erkennen. Durch eine Strategie, die auf gegenseitige Ausgewogenheit bezüglich ausschließbarer und erkennbarer Fehlerklassen baut, kann man viel Aufwand einsparen. Fehler, die ausgeschlossen sind, müssen während des Betriebes nicht mehr erkannt werden (Aufwandsersparung). Bei Fehlern, die während des Betriebes erkannt werden, brauchen aufwendige Tests vor dem Betrieb nicht mehr durchgeführt zu werden (weitere Aufwandseinsparung). Eine erhebliche Aufwandsersparung ergibt sich insbesondere bei gezielter Anwendung von Diversitätsarten, die teilweise automatisch erstellt werden kann.

4 Prozessperipherie

4.1 Schnittstellen zwischen dem technischen Prozess und dem Prozessrechnersystem

4.1.1 Arten von Schnittstellen

Busse

- **Prozessorabhängigkeit** Der prozessorabhängige Bus ist auf einen Prozessor festgelegt. Dadurch ist der Bus einfach, kostengünstig und schnell. Z.B. Ein Systembus zwischen Prozessor und Cache. Der prozessorunabhängige Bus ist für prozessorunabhängige Komponenten, besonders für modulare Steckkartensysteme. Nachteil ist der erhöhte Hardwareaufwand.

- **Gepufferter entkoppelter Bus** Damit langsame Komponenten den Bus nicht blockieren kann einen Teil des Busses über eine Bridge (Fifo-Speicher) entkoppelt werden.

- **Synchroner Bus** Die Datenübertragung erfolgt synchron zum Bustakt. Müssen langsamere Komponenten bedient werden, müssen Wartezyklen eingefügt werden. Es gibt die folgenden Signalleitungen:

  - Adressleitungen $A_0 \dots A_{m-1}$, werden vom Master gesetzt.

  - Datenleitungen $D_0 \dots D_{n-1}$, werden beim Schreiben vom Master gesetzt, beim Lesen vom Slave.

  - R/$\overline{W}$, zeigt an ob gelesen (high) oder geschrieben (low) werden soll.

  - $\overline{READY}$, wird vom Slave auf low gesetzt, wenn er fertig ist.

- **Asynchroner Bus** Die Datenübertragung wird durch Handshakesignale gesteuert. Deswegen gibt es anstelle der $\overline{READY}$ -Leitung beim Asynchronen Bus:

  - $\overline{DTACK}$, Data Acknowledge, Daten vom Slave übernommen, oder auf den Bus gelegt, wenn Signal vom Slave auf low gesetzt.

  - $\overline{AS}$, Adress strobe, Adresse ist gültig, wenn Signal vom Master auf low gesetzt.

  Der Bustakt wird nur für die Signalsynchronisation verwendet. Beim Asynchronen Bus gibt es keine Wartezyklen. Die Zeitspanne zwischen AS und DTACK ist beliebig lang. Asynchrone Busse weisen ein schlechteres Zeitverhalten als synchrone auf.

- **Multimasterfähigkeit** Wenn mehrere Master an einen Bus angeschlossen werden, muss verhindert werden, dass mehrere Master auf einmal den Bus benutzen, dafür gibt es

die Bus-Arbitration. Der Master, der den Bus benutzen möchte, fordert den Bus beim Arbiter an und erhält dann von diesem die Zuteilung. Dieses kann entweder durch einen Master erfolgen oder durch einen externen Arbiter. Der externe Arbiter kann zentral oder dezentral (z.B. über Daisy-Chain oder über einen Identifikationsbus) sein.

- **Busse für Echtzeitsysteme** Für ein Echtzeitsystem ist wichtig:
  - Priorisierung der Master
  - Unterbrechbarkeit von Blocktransfers
  - Festlegung der maximalen Anzahl an Buszyklen in Konkurrenzsituationen
  - Überwachung durch Arbiter
- **PCI-Bus**
  - entkoppelter, gepufferter, prozessorunabhängiger Bus
  - Synchron
  - Daten und Adressen werden gemultiplext
  - 32 oder 64 Bit breit
  - Multimasterfähig, mit zentralem Arbiter
  - Echtzeitfähig (Prioritäten, zeitlich begrenzte Busbelegung)

  Mit Hilfe einer PCI-Bridge kann man unterschiedlich schnelle PCI-Busse hierarchisch kaskadieren. Die BCI-Bridge wird dabei als normales PCI-Device angesprochen. Der PCI-Bus unterstützt zwei verschiedene Transferraten:
  - *Standard*: Es wird erst die Adresse und dann das Datum übertragen. Für das Lesen ist zusätzlich ein Wartezyklus erforderlich. Anzahl Taktzyklen für die Übertragung von n 32 Bit Wörtern: Lesen $3n$ und Schreiben $2n$.
  - *Burst*: Es wird erst eine Adresse und dann dann ein beliebig langer Datenblock übertragen. Für das Lesen ist zusätzlich ein Wartezyklus erforderlich. Anzahl Taktzyklen für die Übertragung von n 32 Bit Wörtern: Lesen $n + 2$ und Schreiben $n + 1$.
- **VME-Bus**
  - Prozessorunabhängiger, multiplexfreier, asynchroner Bus
  - max. Übertragungsrate: ursprünglich 34 MByte/sec, heute bis zu 640 MByte/sec
  - Multiprozessorfähig
  - 32 Bit breit
  - 7 Unterbrechungsebenen, Arbitration Bus
- **Feldbus** Der Feldbus ist ein serielles lokales Bussystem für Industrieumgebungen. Er wird auf der Prozessebene und auf der Feldebene benutzt und ist in der Regel mit der übergeordneten Prozesssteuerung verbunden (Über Feldbuskoppler). Je nach dem in wel-

chem Gebiet der Bus eingesetzt werden soll, sind verschiedene Übertragungseigenschaften erforderlich, dementsprechend gibt es verschiedene Bussysteme: Profibus, CANBus, Interbus(S) und ASi.

- Profibus:
  - Topologie: Linie mit Stichleitungen
  - Buszugriffsverfahren: Master-Slave-Verfahren, Polling
  - Maximale Teilnehmerzahl: 127
  - Übertragungsrate: bis 12 MBit/sec
- CAN-Bus
  - Topologie: Bus, Ring
  - Buszugriffsverfahren: Mulitmaster-Verfahren
  - Maximale Teilnehmerzahl: 2032
  - Übertragungsrate: 1 MBit/sec
  - Übertragungszyklus: Priorisierte Nachrichten
- Interbus(S)
  - Topologie: aktiver Ring
  - Buszugriffsverfahren: Freies Zeitraster, Polling
  - Maximale Teilnehmerzahl: 256
  - Übertragungsrate: 500 kBit/sec
  - Übertragungszyklus: Summenrahmen Telegramm durchläuft alle Teilnehmer
- ASi
  - Topologie: Linie Baum, offener Ring
  - Buszugriffsverfahren: Master-Slave-Verfahren
  - Maximale Teilnehmerzahl: 31 je Segment
  - Übertragungsrate: 150 kBit/sec

4.1.2 ASi – Beispiel für einen Sensor-/Aktor-Bus

ASi ist eine kostengünstige Schnittstelle für binäre Sensoren und Aktuatoren. Das primäre Ziel von ASi ist die Reduzierung des Verkabelungsaufwandes und die Einsparung von E/A-Karten, Klemmleisten, usw.. Die ”Erfinder” von ASi sind elf große Hersteller von binären Aktoren und Sensoren. Diese Unternehmen haben 1990 ein Konsortium gegründet und gemeinsam die Ent-

wicklung von ASi betrieben. ASi heißt Aktuator Sensor Interface und ist für die Vernetzung von Aktoren (Magnetventile, Schütze, etc.) und Sensoren (optisch, induktiv, kapazitiv etc.) konzipiert. ASi st ein Bussystem für die unterste Automatisierungsebene. Mit einem solchen Bus entfällt die aufwendige Parallelverdrahtung zwischen Sensoren und Steuerung. Eine einfache 2-Drahtleitung ermöglicht den Anschluß von bis zu 31 Slaves (max. 124 Sensoren und/oder Aktoren). Der Installationsaufwand ist dadurch deutlich geringer als bei der konventionellen Parallelverdrahtung. Eine weitere Kostenersparnis ergibt sich durch den Verzicht auf Klemmleisten und E/A-Karten.

ASi arbeitet nach dem Master-Slave-Prinzip. Der ASi-Master als Einschubkarte für SPS,Prozessrechner oder als Gateway zu höherem Bussystemen überwacht den Bus und fragt die Teilnehmer zyklisch ab. Verschiedene Koppelmodule machen die meisten konventionellen Sensoren und Aktoren, wie sie heute am Markt verfügbar sind, "busfähig".

Der ASi-Master übernimmt alle Aufgaben, die für die Abwicklung des Busbetriebes notwendig sind. Der Master überwacht und steuert den Bus und entscheidet über den zeitlichen Zugriff auf die angeschlossenen Sensoren/Aktoren. Dazu ruft er die Teilnehmer im System mit ihrer Adresse auf und erwartet eine Antwort. Dieser Zyklus beträgt bei maximalem Ausbau (31 Teilnehmer) 5 ms.

Die an die ASi-Leitung angeschlossenen Sensoren und Akoren müssen die Aufrufe des ASi-Masters verstehen und Daten an diesen übertragen können. Diese Aufgabe übernimmt der ASI-Slave. ASI-Slaves sind passive Teilnehmer und antworten nur, wenn sie vom Master angesprochen werden. Jeder Slave verfügt dazu über eine eigene Adresse. Diese ist bei Auslieferung auf "0" gesetzt. Vor dem Betrieb eines ASI-Systems muss jedem Slave eine gültige Adresse (zwischen 1 und 31) zugewiesen werden. Die Adressierung erfolgt durch ein spezielles Programmiergerät. Die Elektronik des ASi-Slaves ist nahezu komplett in einem hochintegrierten Schaltkreis untergebracht. Der ASi-Slave bietet verschiedene Anwendungsmöglichkeiten:

- Der ASi-Slave wird komplett in den Sensor/Aktor integriert. Über eine spezielle ASi Busklemme wird der Teilnehmer einfach und schnell in das Bussystem eingebunden.

- Koppelmodule sind die einfachste Art, die große Anzahl an verfügbaren Sensoren und Aktoren busfähig zu machen. An die Koppelmodule können einzelne oder bis zu vier konventionelle Sensoren angeschlossen werden. Dazu werden Steckverbinder, wir der weitverbreitete M12-Rundstecker, verwendet. Die Koppelmodule bestehen aus verschiedenen Ober- und Unterteilen, die beliebig miteinander kombiniert werden können. Die Unterteile können aufgeschraubt oder auf 35mm Hutschienen aufgeschnappt werden. Die Oberteile werden mit verschiedenen Ein-/Ausgangangskonfigurationen angeboten. Die Elektronik (Slave-Schaltung) ist im Oberteil untergebracht. Er verfügt über vier Datenleitungen und vier Parameterleitungen. Pro Slave und Zyklus werden über die Datenleitungen 4 Bit Übertragen, die unterschiedlich nutzbar sind, bei Lichtschranken mit integriertem Slave, beispielsweise für den Schalt- und Warnausgang, für die Bereitschaftsanzeige und zur Steuerung des Testeingangs mit dem der Sender kurzzeitig abgeschaltet werden kann. Bei Geräten mit integriertem Slave lassen sich zusätzlich 4 Bit über die Parameterleitung übertragen. Damit können Sensoren oder Aktoren parametriert werden. Alternativ kann ein Slave aber auch über Koppelmodule vier rein binäre Geräte versorgen. Dadurch können bis zu 124 Sensoren/Aktoren (ohne integrierten ASi-Slave) angeschlossen werden. Es kann dann aber jeweils nur ein Ein- bzw. Ausgang (z.B. Schaltausgang) pro Gerät verwendet werden. Die Energieversorgung der angeschlossenen Sensoren/Aktuatoren erfolgt über das Koppelmodul (24 V, bis 100 mA). Endgeräte mit höherem Leistungsbedarf benötigen eine externe Hilfsspannungsversorgung. Diese kann direkt an das Koppelmodul angeschlossen werden. Dafür werden Koppelmodule mit einer zusätzlichen M12-Rundsteckverbindung benötigt.

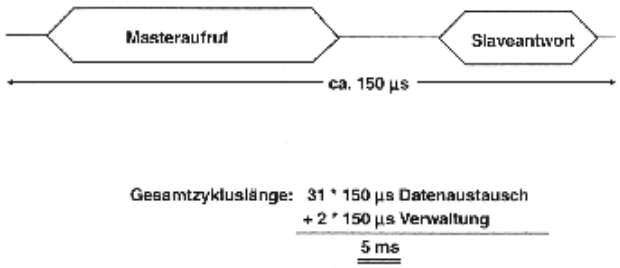


Abbildung 4.1: ASi Master-Slave Kommunikation

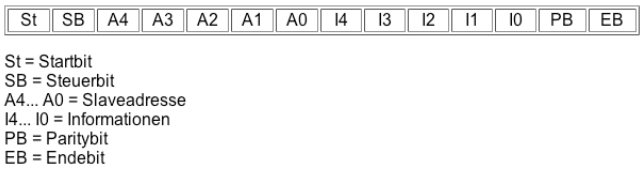


Abbildung 4.2: Telegramm Masteraufruf

ASi verwendet eine ungeschirmte Zweidrahtleitung, auf der Daten und Energie gemeinsam geführt werden. Zur Verdrahtung können Rundkabel verwendet werden. Besondere Vorteile bietet ein ASi-Flachkabel. Durch seine spezielle Form ist eine Fehlverdrahtung ausgeschlossen. Die Kontaktierung erfolgt über Durchdringungstechnik. Das heißt kein Abisolieren, kein Anbringen von Aderendhülsen und somit eine schnellere Montage. ASi verfolgt ein strenges Single-Master-Konzept. Masteraufruf und -antwort des aufzurufenden Teilnehmers wechseln sich innerhalb eines Zyklusses ab. Die Slaves werden der Reihe nach abgefragt. Verfahren zur Steuerung und Überprüfung der Sendeberechtigung können somit entfallen.

- **Masteraufruf:** Das Telegramm des Masteraufrufs besteht aus einem Steuerbit, der gerufenen Slaveadresse, einem Informationsteil und einem Paritybit, sowie Start- und Endebit zur Synchronisation.

- **Slaveaufruf:** Die Antwort besteht aus dem Start-, Parity- und Endebit sowie einem 4 Bit breitem Informationsteil

Beim Kodierungs- und Übertragungsverfahren des ASi werden die zu übermittelnden Daten nach Manchester kodiert, d.h. in der Mitte jeder logischen 0 erfolgt ein Übergang von 1 $\Rightarrow$ 0 und in der Mitte jeder logischen 1 ein Übergang von 0 $\Rightarrow$ 1. Der Vorteil dieser Kodierung liegt darin, dass während der Übertragung eines jeden Bits mindestens ein Übergang auftritt. Somit kann eine Zeitüberwachung zur Fehlererkennung vorgenommen werden. Anschließend wird die zu übertragende Information auf die Bus-Gleichspannung aufmoduliert. Durch die Einprägung eines Stromes durch den Sender entsteht auf dem Bus ein sin^2-förmiges Spannungssignal mit alternierender Pulsfolge. Dieses Verfahren wird Alternierende Puls-Modulation (APM) genannt. Der Empfänger gewinnt aus den Pulsen mit Hilfe zweier Komparatoren das manchesterkodierte

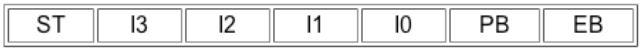


Abbildung 4.3: Telegramm Slaveantwort

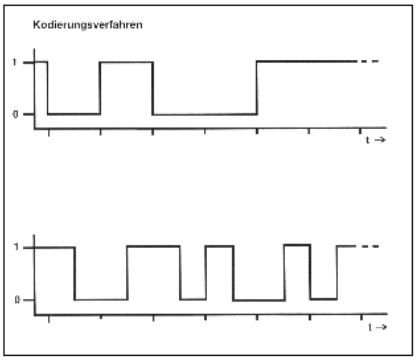


Abbildung 4.4: Kodierung

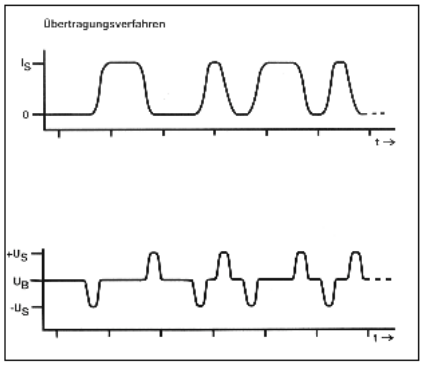


Abbildung 4.5: Übertragung

Signal zurück und kann dieses nun wiederum zum ursprünglichen Datum dekodieren.

4.2 Sensors and Actuators

4.3 Representing the process data in computer systems

4.4 In-/Output of analog signals

Sensors and other data sources usually provide an analog signal. If analog data have to be stored, transmitted, processed or displayed, they are needed in digital form.The subsequent digital processing brings advantages in flexibility and functionality, especially when extensive operations be necessary, for example, calibration, mathematical operations, long-term storage, etc..

We will define analog input by exclusion. Any input that is not digital, that is not defined as two states, (e.g., high/low or one/zero) will be considered analog. Common analog inputs include such measurements as temperature, pressure, flow, strain as well as the direct measurement of voltage and current.

Analog inputs are ”measure” by a device called an A/D (Analog to Digital) converter (sometimes also referred to as an ADC). Though we will discuss A/D converter technology in the next section, it may be useful to mention here that A/D and analog input are often used interchangeably when referring to a data acquistion product. In common usage, an analog input board and A/D board are the same thing. Similarly, an analog input channel and an A/D channel perform the same function.

4.4.1 Analog-Digital-Converters (ADC)

An A/D converter does exactly what its name implies. It is connected to an analog input signal, it measures the analog input and then provides the measurement in digital form suitable for use by a computer. In the process a voltage will be converted into a proportional binary number For

Abbildung 4.6: Maximale Umwandlungszeiten (Stand 1999)

measurement purposes the conversion is carried out mostly linear. For telecom applications, but this can also be weighted (μ -Law, α -Law).

There are four common methods:

- Flash-Converter (1 word per cycle)
- Dual Slope / Integration (1 digit per cycle)
- Successive Approximation (1 Pegel pro Zyklus)
- $\Delta\Sigma$ aka $\Sigma\Delta$ (Overscanning)

Mixed forms such as Half-Flash-Converter, cascade $\Delta\Sigma$-converter are common.

The converters differ in precision (bit width), in conversion-error (linearity), conversion time (max. sample rate), method, effort, output interface and price. Generally speaking, the higher the required conversion accuracy, the greater the necessary conversion time. Furthermore, the faster and more accurate, the more expensive.

Meanwhile, the converters are manufactured almost exclusively in monolithic technology. Hybrid or discrete modules were in the 70s still common, but today they are used only for (very expensive) special applications (eg Burr-Brown).

ADC-characteristics

The characteristic values embody selection criteria. They are determined by the conversion method within certain limits.

- **Resolution**

  The resolution of an A/D input channel describes the number or range of different possible measurements the system is capable of providing. This specification is almost universally provided in term of 'bits', where the resolution is defined as: $2^{\#ofbits} - 1$. For example, 8-bit resolution corresponds to a resolution of one part in $2^8 - 1$ or 255. For resolutions above

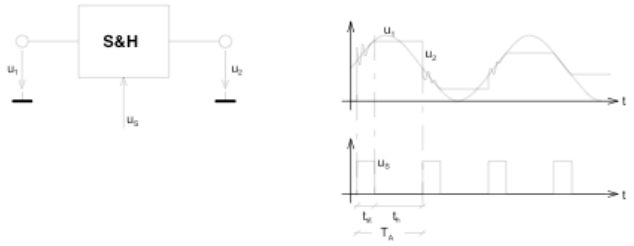


Abbildung 4.7: Sample & Hold-Schaltung mit Darstellung der Takt-, Ein- und Ausgangssignale.

12-bit, the '-1' term becomes virtually insignificant and it is dropped. A resolution of 16-bits corresponds to one part in 2^{16} or 65,536. The minimum difference in a measurement is one bit. This one bit is frequently referred to as the Least Significant Bit or LSB. When combined with an input range, the resolution determines how small a change in the input is detectable. To determine the resolution in engineering units, simply divide the range of the input by the resolution. A 16-bit input with a 0-10 Volt input range provides $10V/2^{16}$ or 152.6 μ Volts.

- **Conversion time** Describes the minimum necessary time required for the converter to proceed.

- **Sample & Hold-circuit** Working as an analog memory the input voltage will be held constant during the conversion. The sampling interval T_A is subdivided into sampling time t_{ST} and holding time t_H. During the sampling time t_{ST} the input signal is captured and stored. During the following holding time t_H , the conversion of the constant voltage into a binary word is performed.

- **Aperture delay** Interval between the application of the hold mode and the actual transition into the hold mode.

- **Aperture jitter** Variation of the aperture delay. The aperture jitter limits the useful bandwidth of the signal to be sampled. The influencef of the jitter leads to an degradation of the signal-to-noise ratio at higher frequencies.

- **Input Offset** Assuming all other errors are zero, input offset is a constant difference between the measured input and the actual input voltage. For example, if the input offset voltage was +0.1 volt, measurements of perfect 1, 2 and 5-volt input signals would provide readings of 1.1, 2.1 and 5.1 volts, respectively. In a real system, the other errors are never zero, which complicates the measurement of input offset. Most analog input specifications define the input offset as the measurement error at 0 volts.

  Some products provide an 'auto zeroing' capability. This function drives the input offset error to zero, or at least to a level low enough that its contribution is no longer significant relative to other errors.

- **Gain Error** It is easiest to illustrate this error by first assuming all other errors are zero. Gain error is the differ- ence in the slope (in volts per bit) between the actual system and an ideal system. For example, if the gain error is 1%, the gain error at 1 volt would be 10 millivolts (1 * .01), while the error at 10 volts would be ten times as large at 100 millivolts. In a real world system where the other errors are not zero, the gain error is usually defined as the error of the measurement as a percentage of the full scale reading. For example, in our 0 - 10 volt example range, if the error at 10 V (or more often at a reading arbitrarily close

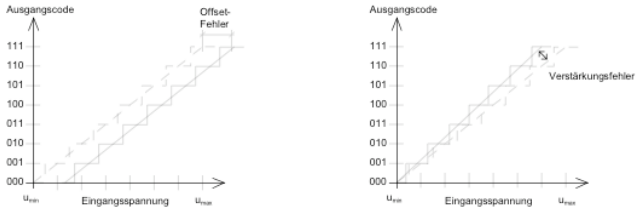


Abbildung 4.8: Einfluss der Offset- und Verstärkungsfehler beim AD-Wandler.

to 10 volts such as 9.99 V) is 1 millivolt, the gain error specified would be $100 * (.001/10)$ or .01%. For higher precision measurement systems, the gain error is often specified in parts per million (or ppm) rather than percent as itś a bit easier to read. To calculate the error in parts per million, simply multiply the input error divided by the input range by one million. In our example above, the 0.01% would be equivalent to $1,000,000 * .001/10$ or 100 ppm. Though many products offer auto-calibration, which substantially reduces the gain error, it is not possible to eliminate it completely. The automated gain calibration is almost always performed relative to an internally supplied reference voltage. The reference voltage will drift over time and any error in the reference will translate into a gain error. It is possible to create references with arbitrarily small errors. However, as the gain error gets small relative to other system errors, it becomes economically unfeasible to improve the reference accuracy. In addition to the cost penalty involved in providing the 'pseudo perfect' reference, one of the errors, if not the largest, in most references is the drift with temperature. The only way to eliminate this drift is to maintain the reference temperature at a constant level. This is not only expensive, but it also requires a significant amount of power, which increases overall system power consumption.

- **Non-Linearity** As its name implies, non-linearity is the difference between the graph of the input measurement versus actual voltage and the straight line of an ideal measurement. The non-linearity error is composed of two components, integral non-linearity (INL) and differential non linearity (DNL).

  Of the two, integral non-linearity is typically the specification of importance in most DAQ systems. The INL specification is commonly provided in 'bits' and describes the maximum error contribution due to the deviation of the voltage versus reading curve from a straight line. Though a somewhat difficult concept to describe textually, INL is easily described graphically and is depicted in the next figure. Depending on the type of AD converter used, the INL specification can range from less than 1 LSB to many, or even tens, of LSBs.

  Differential non-linearity describes the 'jitter' between the input voltage differential required for the AD converter to increase (or decrease) by one bit. The output of an ideal AD converter will increment (or decrement) one LSB each time the input voltage increases (or decreases) by an amount exactly equal to the system resolution. For example, in a 24-bit system with a 10-volt input range, the resolution per bit is $0.596 \mu volt$. Real AD converters, however, are not ideal and the voltage change required to increase or decrease the digital output varies. DNL is typically ±1 LSB or less. A DNL specification greater than ±1 LSB indicates it is possible for there to be 'missing' codes. Though not as problematic as a non-monotonic DA converter, AD missing codes do compromise measurement accuracy.

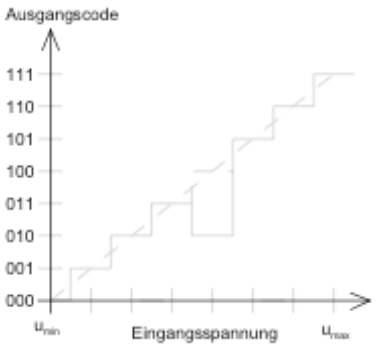


Abbildung 4.9: Monotoniefehler beim AD-Wandler. Sie äussern sich in "Missing Codes".

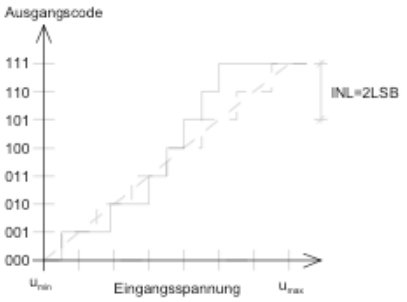


Abbildung 4.10: Integrale Nichtlinearität als Abweichung zwischen tatsächlichem und idealem Wert.

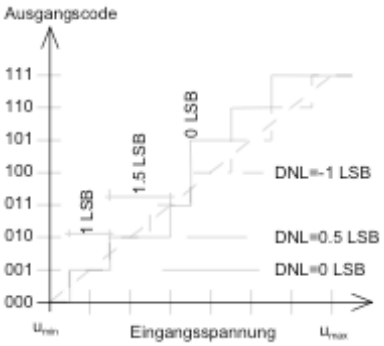


Abbildung 4.11: Differenzielle Nichtlinearität als Auswirkung der nicht äquidistanten Quantisierungsstufenbreite.

Nyquist-Abtastung

Erfolgt die Abtastung eines Analogsignals f_A mit einer Abtastfrequenz $f_S > 2f_A$, spricht man von Nyquist-Abtastung. Mathematisch gesehen, entspricht die Abtastung einer Eingangsspannung $u_1(t)$ einer Faltung mit dem periodischen Dirac-Puls der Periode T_S.

$$\tilde{u}_1(t) = T_S \sum_{k=0}^{\infty} u_1(kT_S)\delta(t - kT_S) \tag{4.1}$$

Wir erhalten eine Impulsfolge in der Art wie in Bild 4.12 dargestellt mit den Werten $u_1(0), u_1(T_S), u_2(2T_S), \ldots$ Bereits ohne grosse mathematische Analyse lässt sich vermuten, dass durch die steilflankigen Impulse das Frequenzspektrum des ursprünglichen Eingangssignals verändert wird, indem höherfrequente Anteile zugefügt werden. Das Spektrum des abgetasteten Signals wird mit einer Fourier-Transformation der abgetasteten Spannung $\tilde{u}_1$ zu:

$$\tilde{X}(jf) = T_S \sum_{k=0}^{\infty} u_1(kT_S)e^{-2kj\pi fT_S} \tag{4.2}$$

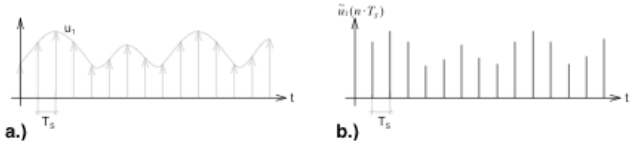


Abbildung 4.12: Abtastung eines analogen Eingangssignals mit einer periodischen Folge von Diracstössen und resultierenden Impulsfolge.

Man erkennt das periodische Spektrum, wobei die Periodizität gleich der Abtastfrequenz f_S ist. Ferner ist das Spektrum der abgetasteten Funktion $|\tilde{X}(jf)|$ im Bereich $-\frac{1}{2}f_s \leq f \leq \frac{1}{2}f$ identisch mit dem Spektrum des Eingangssignales $|X(jf)|$. Dies ist insofern bemerkenswert, weil meist nur wenige Abtastwerte zur Bestimmung benutzt werden. Das Orginalspektrum erscheint solange

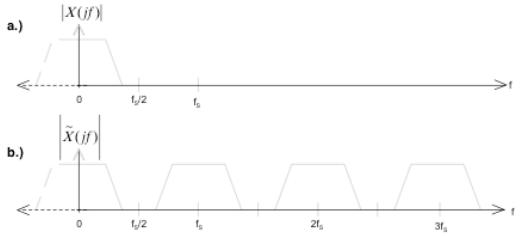


Abbildung 4.13: Spektrenduplizierung bei der Abtastung: Eingangsspektrum und periodisches Ausgangsspektrum nach der Abtastung.

unverändert wie sich die periodisch wiederkehrenden Spektren nicht überlappen. Überlappen sich die Spektren, treten sog. Aliasing-Effekte auf.

Aus obigen Forderungen ergibt sich das Nyquist-Kriterium für die Abtastung:

$$f > 2f_{max} \tag{4.3}$$

Daher kann ein abgetastetes Signal vollständig in Amplitude und Phase rekonstruiert werden, wenn mehr als zwei Abtastwerte pro Periode vorliegen. Die Gleichheit gilt übrigens in der obigen Formel nicht!

In der Praxis kann die Abtastung nicht mit einer idealen Dirac-Funktion erfolgen, sondern mit einem Rechteckfenster schmaler Breite. Dies wirkt aber wie ein Tiefpassfilter, indem das Spektrum mit einer si-Funktion gewichtet wird.

$$\tilde{X}^*(jf) = \frac{sin(\pi f T_S)}{\pi f T_S} T_S \sum_{k=0}^{\infty} u_1(kT_S) e^{-2kj\pi f T_S} = \frac{sin(\pi f T_S)}{\pi f T_S} \tilde{X}(jf) \tag{4.4}$$

Bei der halben Abtastfrequenz tritt durch die si-Funktion eine Abschwächung um den Faktor 0.64 auf. Beim Entwurf von Filtern ist diese Verzerrung zu berücksichtigen und wird bei Filtersyntheseprogrammen meist automatisch durchgeführt. **Zusammenfassung:** Bei Nyquist-Abtastung erscheint das Spektrum des Eingangssignals jeweils alle f_S gespiegelt. Das hat zur Konsequenz, dass in ein System mit Nyquist-Abtastung nur Eingangsfrequenzen bis $f_{S/2}$ eingespiesen werden können, sonst treten Aliasing-Effekte auf. Durch ein analoges Tiefpassfilter vor dem Abtaster wird sichergestellt, dass die Eingangsfrequenzen über $f_{S/2}$ genügend gedämpft werden. Ist das Eingangssignal bereits genügend bandbreitenbegrenzt kann das Filter auch entfallen. Durch die nicht ideale Abtastung erscheint das Spektrum mit einer si-Funktion gewichtet, d.h. spektral höhere Frequenzen werden abgeschwächt. Richtwert: 64% bei $f_{S/2}$.

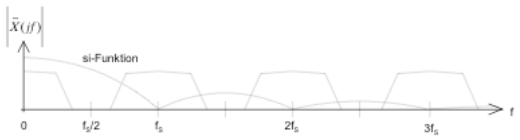


Abbildung 4.14: Si-bewertetes Ausgangsspektrum nach der Abtastung mit einem nicht-Dirac-Abtaster.

Abbildung 4.15: Prinzip übertastender AD-Wandler

Übertastverfahren

Zur Reduktion des analogen Filteraufwandes und zur Erhöhung der Wandlungsgenauigkeit werden vielfach Überabtastverfahren eingesetzt. Dabei wird das Eingangssignal mit einem ganzzahligen Vielfachen L der Abtastfrequenz abgetastet und gewandelt. Am Ausgang erfolgt eine Abtastratenwandlung (Dezimierung um L) auf die gewünschte Abtastrate. **Vorteile:**Wegen der spektralen Gleichverteilung der Fehlerleistung zwischen 0 und der Abtastfrequenz f_S kann bei einem Überabtastverfahren mit $L \cdot f_S$ im Nutzband um den Faktor L abgesenkt werden. Bei gleich grossen Quantisierungsstufen Q wird die spektrale Leistungsdichte S des Quantisierungsfehlers: Im Nutzband $[-f_{A'}, +f_A]$ wird bei Einfach-Abtastung mit $f_S = 2f_A$ und dem Leistungsdichtespektrum $S_{ee}(f) = \frac{Q^2}{12 f_s}$ die Fehlerleistung:

$$N_A^2 = \sigma_E^2 = 2 \int_0^{f_A} S_{ee}(f)\, df = \frac{Q^2}{12} \tag{4.5}$$

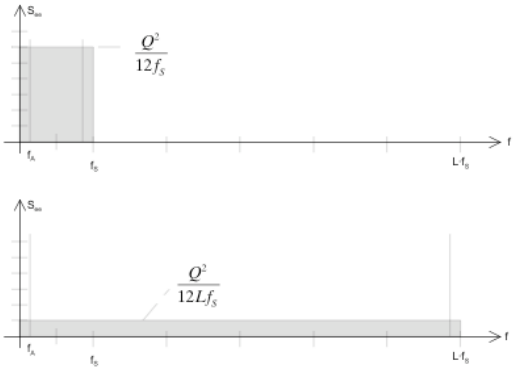


Abbildung 4.16: Spektrale Leistungsdichte des Quantisierungsfehlers Q bei einfacher Abtastung und L-facher Überabtastung

Bei Überabtastung erfolgt eine Reduktion der spektralen Leistungsdichte des Quantisierungsfehlers im Nutzband um den Faktor $\frac{1}{L}$. Dies ergibt eine Fehlerleistung:

$$N_A^2 = \sigma_E^2 = \frac{Q^2}{12} \cdot \frac{1}{L} \tag{4.6}$$

Da bei Überabtastung der Frequenzbereich des analogen Filters am Eingang bis $L \cdot f_S - \frac{f_S}{2}$ reicht, können einfachere Filter verwendet werden. Nach erfolgter Wandlung werden mit einem digitalen Filter alle Frequenzen $> f_S/2$ abgefiltert. Der Abtastratenwandler reduziert die Abtastrate um den Faktor L, indem jeder L-te Wert aus dem digitalen Tiefpassfilter übernommen wird.

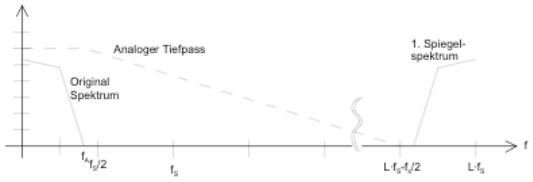


Abbildung 4.17: Anforderung an den analogen Tiefpass und Spektrum bei L-facher Abtastung.

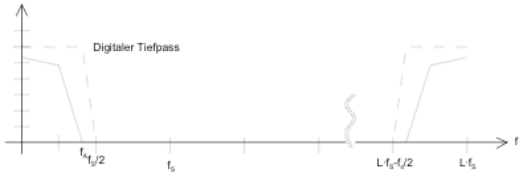


Abbildung 4.18: Filterung mit digitalem Tiefpass nach AD-Wandlung bei L-facher Abtastrate.

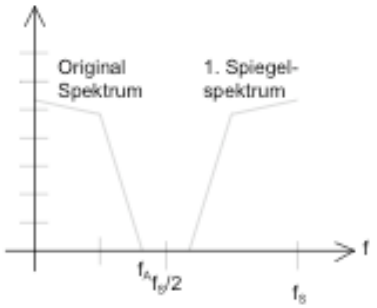


Abbildung 4.19: Spektrum nach erfolgter Abtastratenreduktion.

Sample&Hold Schaltungen

Sample und Hold-Schaltungen sind integraler Bestandteil der AD-Wandler. Aufgabe der S&H Schaltungen ist die Eingangsspannung durch Analogspeicherung während des Umsetzprozesses konstant zu halten. Würde der AD-Wandler ohne S&H-Glied betrieben, könnten nur Eingangsspannungen mit geringen Anstiegsgeschwindigkeiten umgesetzt werden.

Die schematische Ausführung eines Sample&Hold Gliedes: Während der Schalter S geschlos-

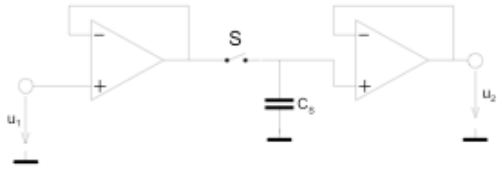


Abbildung 4.20: Funktioneller Aufbau einer Sample&Hold Schaltung.

sen wird, erfolgt die Ladung des Kondensators auf den Wert der Eingangsspannung. Wird S geöffnet, wird der Wert der Kondensatorspannung gehalten. Die beiden OpAmps dienen zur Entkopplung, damit der Kondensator einerseits möglichst rasch der Eingangsspannung folgt, und andererseits die Ladung im Kondensator im Haltezustand lange erhalten bleibt und zwar unabhängig von der Belastung.

Zu den nicht idealen Eigenschaften der Sample&Hold-Schaltungen zählt die beschränkte Anstiegsgeschwindigkeit. Sie wird hauptsächlich durch den Maximalstrom des Impedanzwandlers am Eingang bestimmt. Dann folgt ein Einschwingvorgang, der durch die parasitären Widerstände des Schalters und vom Ausgangswiderstand des Impedanzwandlers bestimmt wird. Nach der sog. Einstellzeit T_E hat die Ausgangsspannung den Wert der Eingangsspannung mit einer definierten Toleranz erreicht. Wenn der Schalter S geöffnet wird, erfolgt der Wechsel in den Haltezustand. Der Schalter braucht dafür eine bestimmte Zeit zum Öffnen, die sog. Aperturzeit t_{apu}. Sie ist meist nicht konstant sondern schwankt in Abhängigkeit von der Eingangsspannung. Die Schwankung wird als Apertur-Jitter bezeichnet.

Die wichtigste Größe im Haltezustand ist die Haltedrift (Droop). Sie beschreibt, um wieviel sich die Ausgangsspannung durch Selbstentladung verändert. Diese Größe wird durch das Dielektrikum des Speicherkondensators, den Sperrstrom des Schalters und den Eingangsstrom des Impedanzwandlers bestimmt.

S&H-Schaltungen werden mittlerweile ausschliesslich als IC eingesetzt. Eine Zusammenstellung gängige S&H-Schaltung nach [17] ist: Da für eine fehlerfreie Quantisierung die Spannungsänderung während der Umsetzung kleiner als eine Quantisierungsstufe sein muss, wird die maximale Eingangsspannungsänderung für ein N-Bit Wandler allgemein:

$$\frac{dU}{dt}|_{max} = 2^{-N}\frac{u_{max}}{t_{con}} \tag{4.7}$$

u_{max} : Max. Eingangsspannung[V] t_{con} : Umsetzzeit[s] N : Wortbreite des Wandlers [Bit]

So wird für einen 12Bit AD-Wandler (1/4096 Auflösung) ohne S & H-Schaltung mit einem $u_{max} = 10V$ und einer Umsetzzeit $t = 0.1s$ die maximal zulässige Eingangsspannungsänderung:

$$\frac{dU}{dt}|_{max} = 2^{-12}\frac{10}{0.1} = 0.0244\frac{V}{s} \tag{4.8}$$

Wird hingegen eine Sample&Hold-Schaltung verwendet, wird die Umsetzzeit des AD-Wandlers bedeutungslos. An ihre Stelle tritt die Apertur-Jitterzeit t_{apu}. Unter Verwendung eines S&H-Gliedes mit einer Apertur-Jitterzeit von $t = 3ns$ wird die maximal zulässige Eingangsspannungs"nderung:

$$\frac{dU}{dt}|_{max} = 2^{-N}\frac{u_{max}}{t_{apu}} = 2^{-12}\frac{10}{3\cdot 10^{-9}} = 813.8\frac{kV}{s} = 0.83138\frac{V}{s} \tag{4.9}$$

Typ	Hersteller	Speicher-kondensator	Einstellzeit	Genauigkeit	Max. Anstiegsgeschw.	Haltedrift	Technologie
LF398	viele	10nF	20us	10Bit	0.5V/us	3mV/s	BiFET
LF398	viele	1nF	4us	10Bit	5V/us	30mV/s	BiFET
AD585	Analog Devices	100pF (intern)	3us	12Bit	10V/us	100mV/s	Bipolar
SHC5320	Burr-Brown	100pF(intern)	1.5us	12Bit	45V/us	100mV/s	Bipolar
SHM20	Datel	intern	1us	12Bit	45V/us	100mV/s	Bipolar
CS3112	Crystal	intern	1us	12Bit	4V/us	1mV/s	CMOS
CS31412[4]	Crystal	intern	1us	12Bit	4V/us	1mV/s	CMOS
AD781	Analog Devices	intern	0.6us	12Bit	60V/us	10mV/s	BIMOS
AD682[2]	Analog Devices	intern	0.6us	12Bit	60V/us	10mV/s	BIMOS
AD684[4]	Analog Devices	intern	0.6us	12Bit	60V/us	10mV/s	BIMOS
HA5330	Harris	90pF(intern)	0.5us	12Bit	90V/us	10mV/s	Bipolar
AD783	Analog Devices	intern	0.2us	12Bit	50V/us	20mV/s	BIMOS
LF6197	National	10pF(intern)	0.2us	12Bit	145V/us	0.6V/s	BiFET
HA5330	Harris	intern	50ns	12Bit	130V/us	100V/s	Bipolar
AD9100	Analog Devices	22pF(intern)	16ns	12Bit	850V/us	1kV/s	Bipolar
SHM12	Datel	15pF(intern)	15ns	12Bit	350V	500V/s	Bipolar
AD9101	Analog Devices	intern	7ns	10Bit	1.8kV/us	5kV/s	Bipolar
SHC702	Burr-Brown	intern	0.5us	16Bit	150V/us	0.2V/s	Hybrid
SP9760	Sipex	intern	0.35us	16Bit	120V/us	1V/us	Hybrid
SHC803	Burr-Brown	intern	0.25us	12Bit	160V/us	0.5V/s	Hybrid
SHC49	Datel	intern	0.16us	12Bit	300V/us	0.5V/s	Hybrid
HS9730	Sipex	intern	0.12us	12Bit	200V/us	50V/s	Hybrid
SHM43	Datel	intern	35ns	12Bit	250V/us	1V/s	Hybrid
CL942	Comlinear	intern	25ns	12Bit	300V/us	20V/s	Hybrid
SHC601	Burr-Brown	intern	12ns	10Bit	350V/us	20V/s	Hybrid
HTS0010	Analog Devices	intern	10ns	8Bit	300V/us	50V/s	Hybrid
CL940	Comlinear	intern	10ns	8Bit	500V/us	20V/s	Hybrid

[2]: Zweifach-S&H [4]: Vierfach-S&H

Abbildung 4.21: Gängige Sample&Hold Schaltungen.

Obwohl auch das zweite Resultat nicht berauschend ist, erkennt man aus diesem Beispiel die Notwendigkeit einer Sample&Hold-Schaltung für AD-Wandler und kann abschätzen, welche Anforderungen an den Apertur-Jitter gestellt werden müssen. Ebenso hat die Slew-Rate der S&H - Schaltung einen Einfluss und muss berücksichtigt werden.

Flash converter

Flash converters can be extremely fast, though they do not typically offer high resolution. Sample rates in the gigahertz range are possible, but flash converters are rarely seen with resolutions greater than 8 bits. A flash converter is really nothing more than a string of comparators with highly trimmed input resistors. One side of each comparator input is set at one bit of resolution greater than the next. That means any given input to the AD converter will always fall between the reference inputs of two adjacent comparators, driving one comparator output high, with the adjacent output low. This data is then taken into a decoder, and the position where the switch from 0 to 1 occurs defines the AD output. As mentioned previously, it is a simple concept, but as a Flash converter requires at 2^n comparators, the amount of circuitry required on the chip gets huge as the converter resolution increases. An 8-bit flash converter requires 255 comparators. Thatá lot of analog circuitry on a single chip, even at todayś densities. Try to add one bit of resolution and you push the total number of comparators required to 511. A very difficult challenge indeed.

Kaskaden-Wandler

Man kann den Aufwand des Parallelverfahrens für größere Wortbreiten reduzieren, wenn Kompromisse in Umsetzgeschwindigkeit eingegangen werden. Umsetzraten von > $100MHz$

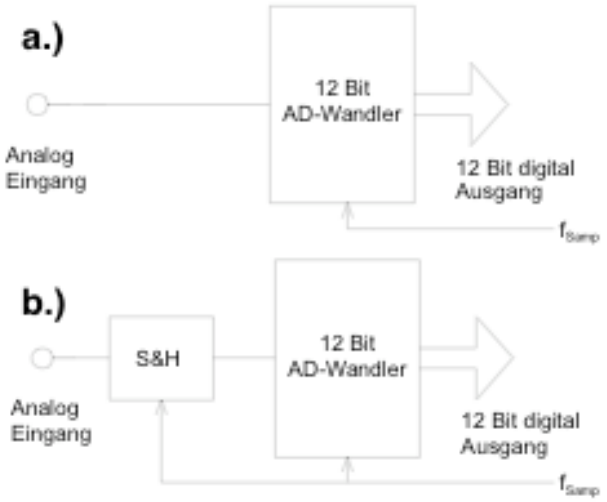


Abbildung 4.22: AD-Wandler mit und ohne Halteglied

bei 10Bit sind aber problemlos erreichbar. Beim Kaskadenprinzip in zwei m-Bit Wandlungen, wobei N=2m. Es erfolgt zuerst eine "Grobdigitalisierung" der höherwertigen m Bits. Anschliessend werden die niederwertigen m Bits digitalisiert. Intern werden für beide Aktionen Parallel-Wandler benutzt. So kann ein 10Bit Wandler durch eine Kaskade zweier 5Bit Parallel-Wandler realisiert werden. Dabei wird in einem ersten Schritt eine Grobquantisierung mit 5 Bit durchgeführt. Ein DA-Wandler bildet die zu diesem effektiv gehörende Spannung. In einem Subtrahierer wird die Differenz der noch zu digitalisierenden Spannung gebildet. Dies erfolgt in einem zweiten 5 Bit Wandler. Allerdings sind bei diesem Verfahren die Anforderung an den ersten AD-Wandler sehr hoch. Er muss über volle Umsetzgenauigkeit (d.h. 10Bit Präzision) verfügen, obwohl er nicht volle Wortbreite hat. Sonst wird die gebildete Differenz falsch und der zweite AD-Wandler wird übersteuert. Dies äussert sich in Missing Codes. Eine digitale Fehlerkorrektur mit erhöhtem Schaltungsaufwand ist in Grenzen möglich, vgl. hierzu [17] S.1051.

Beim Subranging-Prinzip erfolgt auch eine Aufteilung in zwei m-Bit Wandlungen. Die Wandlung beider Gruppen erfolgt aber sequentiell, wobei nur ein AD-Wandler benötigt wird. Die Umsetzzeit steigt aber gegenüber dem Half-Flash-Kaskadenwandler an. Im ersten Zyklus steht der Schalter in Stellung H. Es erfolgt die Wandlung der höherwertigen m Bits. Im zweiten Schritt steht der Schalter auf L. Am AD-Wandler steht die um 2^m verstärkte Differenzspannung an. Diese bestimmt die niederwertigen m Bits. Die Ablaufsteuerung erfolgt durch die Logik die mit der Abtastfrequenz getaktet wird.

Successive Approximation (SA)

They typically provide resolutions in the 10 to 18-bit range, and depending on the resolution, offer sample rates up to tens of Megasamples per second. The basic underlying technology of SA involves comparing the input to the output of an on-chip D/A converter. Based upon whether the D/A converter output is higher or lower than the input, the D/A converter output is raised or lowered. In this manner, the SA converter ultimately zeroes in and when the D/A converter output is equal to the input (within one LSB), the iteration ends and the current digital word is written to the A/D output.

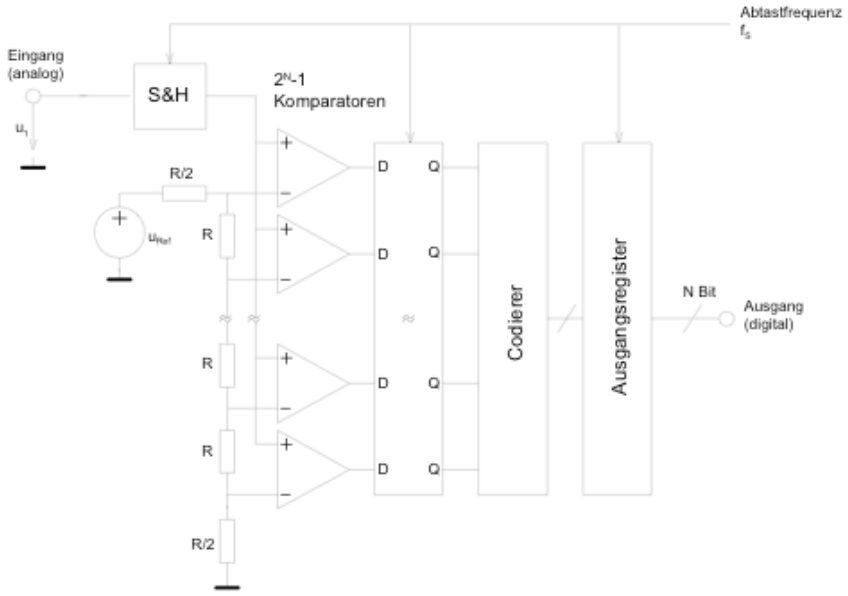


Abbildung 4.23: Prinzip-Blockschaltbild eines AD-Wandlers nach dem Parallelverfahren.

Das Verfahren kann mit einer binären Suche in maximal N Zyklen den Wert bestimmen. Dabei wird mit dem höchstwertigen Bit begonnen und geprüft, ob die Eingangsspannung größer oder kleiner ist. Ist sie größer, wird das Bit beibehalten. Ist sie kleiner, wird das Bit Null gesetzt. Diese Prozedur wird für jede binäre Stelle wiederholt. Der Wandler approximiert schrittweise den Wert in N Zyklen:

Zählverfahren

AD-Wandler nach dem Zählverfahren haben den geringsten Schaltungsaufwand, sind aber relativ langsam. Die Wandlungszeiten liegen im Bereich 1ms. . . 1s, je nach Auflösung und Abtastfrequenz. Daher werden diese Wandlertypen vor allem zur Messung bei langsam ändernden Spannungen eingesetzt, wie z.B. in Digitalvoltmetern und Temperaturmessern. Unter dem Begriff "Zählverfahren-Wandler" werden verschiedene Typen zusammengefasst.

- Tracking (Nachlaufverfahren)
- Single Slope (Ein-Rampen Verfahren)
- Dual Slope (Zwei Rampen Verfahren), hat die größte Verbreitung
- **Tracking Verfahren** Das Nachlaufverfahren arbeitet ähnlich dem SAR-Verfahren jedoch wird der digitale Wert mit einem gesteuerten Vor-Rückwärts-Zähler erzeugt. Da jeweils in einem Taktzyklus der Zähler nur um eine Stelle bewegt wird, erfolgt die Wandlung wesentlich langsamer als bei einem SAR-Wandler. Vorteil der Tracking-Wandler ist der geringe Schaltungsaufwand. Da heute praktisch ausschliesslich mit IC-Wandler gearbeitet wird, fällt dieser Vorteil weg.
- **Dual-Slope Verfahren** Beim Dual-Slope Verfahren wird in einer ersten Phase das Eingangssignal eine definierte Zeit integriert (geladen). In der zweiten Phase erfolgt eine

Typ	Hersteller	Abtast-frequenz	Betriebs-spannung	Verlust-leistung	Eing. Kapazität	Apertur-Jitter	Logik Familie
8Bit							
ADC307	Datel	125MHz	-5.2V	870mW			ECL
ADC309	Datel	500MHz	-5.2V	2800mW	6pF	11ps	ECL
HI306	Harris	140MHz	+5V	360mW	21pF	10ps	TTL
MAX104	Maxim	1000MHz	±5V	3500mW			ECL
MAX1114	Maxim	150MHz	-5.2V	2200mW	10pF	5ps	ECL
MAX1150	Maxim	500MHz	-5.2V	5500mW	15pF	2ps	ECL
MAX1151	Maxim	750MHz	-5.2V	5500mW	15pF	2ps	ECL
TDA7818	Philips	600MHz	-5.2V	990mW	5pF		ECL
TDA8793	Philips	100MHz	+3.3V	150mW	2pF		CMOS
CXA1276	Sony	500MHz	-5.2V	2800mW			ECL
SPT7710	CPT	150MHz	-5.2V	2200mW	10pF	5ps	ECL
SPT7750	CPT	500MHz	-5.2V	5500mW	15pF	2ps	ECL
SPT7760	CPT	1000MHz	-5.2V	5500mW	15pF	2ps	ECL
10Bit							
AD9020	Analog Devices	60MHz	±5V	2800mW	45pF	5ps	TTL
AD9060	Analog Devices	75MHz	±5V	2800mW	45pF	5ps	ECL
TDA8762	Philips	80MHz	+5V	380mW	5pF		CMOS

Abbildung 4.24: Beispiele für handelsübliche Parallel-AD-Wandler sind nach [17]

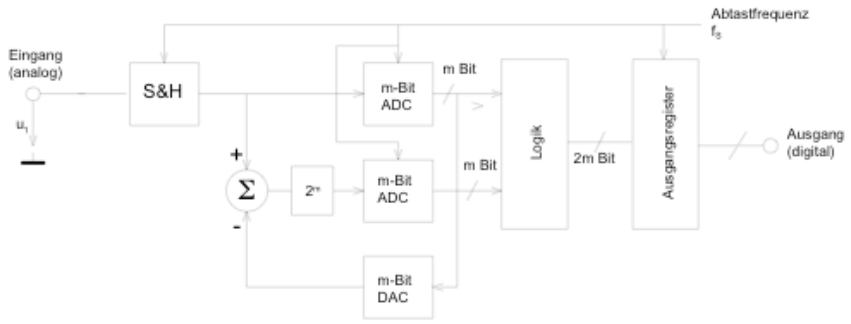


Abbildung 4.25: Blockschaltbild eines KaskadenAD-Wandlers (Half-Flash).

definierte Rückintegration (entladen) mit der Referenzspannung. Aus der Anzahl Takte, die zum Erreichen des Nullwertes gebraucht wurden, wird in einem Zähler der digitale Wert gebildet. Dies ist der Wert, der aus dem Verhältnis gebildet wird:

$$\frac{U_{1S}}{U_{Ref}} = \frac{t_2}{t_1} \tag{4.10}$$

Den Verlauf der Spannung am Integratorausgang für verschiedene Eingangsspannungen u_{1A}, u_{1B}, u_{1C} zeigt das folgende Bild. Man beachte, dass die Steigung der Entladekurve immer gleich ist, für unterschiedliche Eingangsspannung die Ladekurve aber unterschiedliche Steigung aufweist.

- **Delta-Sigma converter** Some manufacturers refer to this type of converter as ΔΣ, while others call it a ΣΔ. The delta sigma converter is rapidly becoming the standard by which other converters are judged and is seen in more and more data acquisition devices each day. Perhaps the most differentiating feature of sigma delta converters is that they provide resolution up to 24-bits. Another interesting feature of converters is that they inherently trade off sample rate with resolution. Many converters provide an ability to sample at high speeds at low(er) resolutions or to slow down and sample at a rate that provides the maximum possible resolution. Without going too much into the technology, the concept is fairly simple. A lower resolution converter inside the converter (typically 1-bit) is drama-

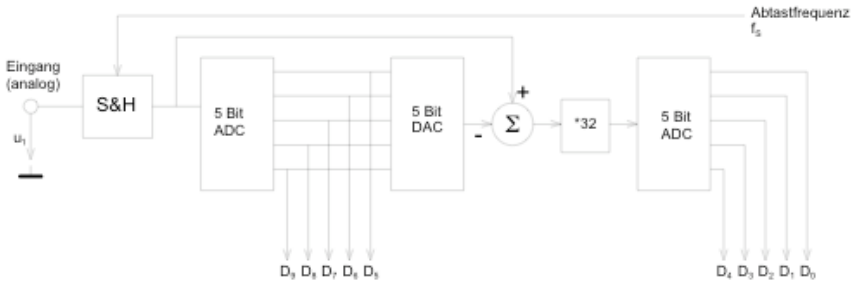


Abbildung 4.26: Beispiel eines 10-Bit Half-FlashWandlers mit zwei 5-Bit Umsetzer.

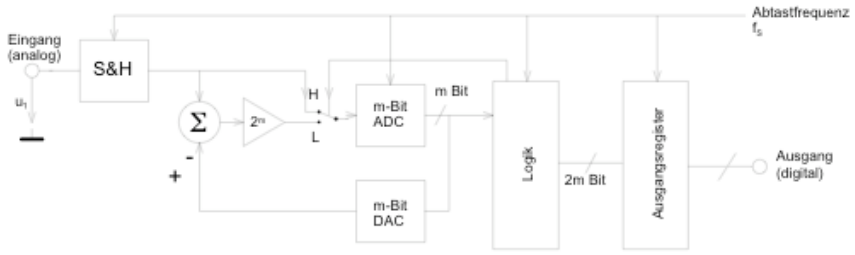


Abbildung 4.27: Blockschaltbild Subranging Wandler.

tically oversampled, i.e., sampled faster (in this case, orders of magnitude faster) than the sample rate your data acquisition system desires.

The large number of ”one-bit” A/D conversions is then digitally integrated. The resolution is determined by the number of ”internal” samples taken. A high number of internal samples provides a higher resolution with a lower overall converter sample rate. If you sample fewer times, you receive your A/D converter data faster, but at lower resolution. The converters are not without their share of design issues, but these are being addressed by the various manufacturers at a high rate and they will almost certainly supplant the Successive Approximation converter in the reason-ably near future.

4.4.2 Digital-Analog-Wandler (DAC)

Sie wandeln einen digitalen Eingangswert in einen proportionalen analogen Ausgangswert um. Die Wandlung erfolgt bis auf das $\Delta\Sigma$-Verfahren ausschliesslich in paralleler Form. Daher sind die erreichbaren Wandlungsraten hoch.

AD-Wandler-Kenngrößen

Die Definition der Kenngrössen erfolgt anlog den Aussagen zum AD-Wandler. Eine zusammenfassende Beschreibung der Kenngrössen wird nach [18]:

- **Auflösung** Sie besagt, welche Wortbreite N in Bits zur Umsetzung verwendet wird.

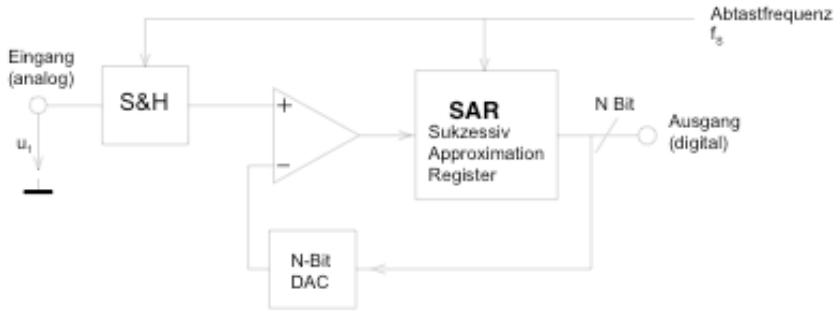


Abbildung 4.28: Blockschaltbild eines Wandlers nach dem Verfahren der Sukzessiven Approximation.

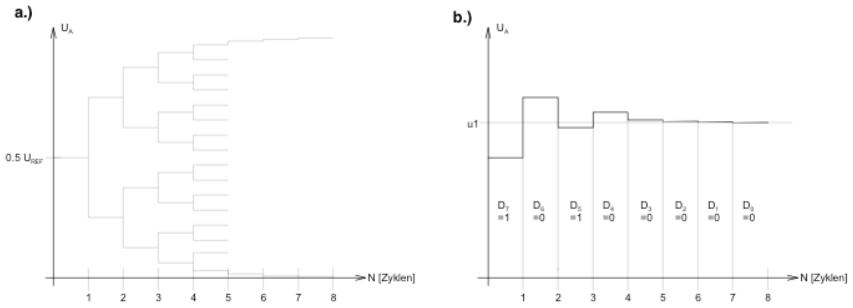


Abbildung 4.29: Sukzessive Approximation nach binärem Baum. Der Ausgangswert wird nach spätestens N Zyklen erreicht und Beispiel zur Entwicklung des Ausgangswertes beim SAR-Wandler.

- **Einstellzeit (Settling Time)** Sie beschreibt die maximal notwendige Zeit t_{SE}, vom Anlegen des binären Wortes am Eingang bis zum Erreichen des anlogen Ausgangsendwertes innerhalb einer Toleranz. Sie ist daher als Umsetzzeit zu sehen. Die Einstellzeit definiert die maximale Wandlungsfrequenz mit $f_{max} = 1/t_{SE}$.

- **Offset- und Verstärkungsfehler** Offsetfehler äussern sich in einer seitlichen Verschiebung der Umsetzkennlinie. Verstärkungsfehler in einer Abweichung der idealen Steigung. Meist kann schaltungstechnisch eine Kompensation erfolgen.

- **Integrale Nichtlinearität (INL)** Sie beschreibt den Fehler zwischen der realen Ausgangsspannung und dem idealen Wert. Er wird in Anzahl LSB angegeben.

- **Differenzielle Nichtlinearität (DNL)** Idealerweise bewirkt eine Erhöhung des Codewortes um ein LSB eine Erhöhung der Ausgangsspannung um eine Quantisierungsstufe. Die differenzielle Nichtlinearität (DNL) beschreibt den maximalen Stufenbreitenfehler in Anzahl LSB.

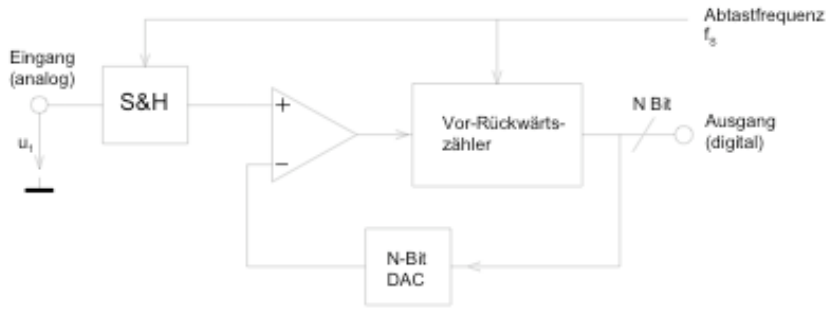


Abbildung 4.30: Blockschaltbild eines AD-Wandlers nach dem Tracking-Verfahren.

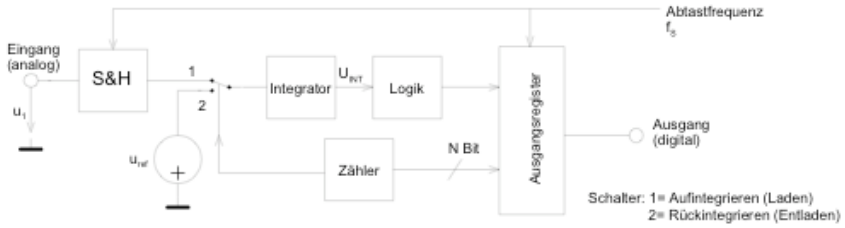


Abbildung 4.31: Blockschaltbild Dual-Slope Wandler.

4.5 Ein-/Ausgabe von binären und digitalen Signalen

An den digitalen Ein- und Ausgängen liegen Binärsignale mit folgenden Signalniveaus an.

- TTL-Signake (im Bereich von 0 bis 5V)
- 24V Gleichspannung (z.B. bei Relais)
- 230 V Wechselspannung (z.B. bei Schützen)

Vorzugsweise wird mit TTL-Pegel gearbeitet. Dabei müssen ”benachbarte” Bits nicht unbedingt etwas miteinander zu tun haben: ein Bit meldet beispielsweise ”Förderband 4 ist eingeschaltet”, das Nachbarbit dagegen ”Ventil in Rohrleitung 17 ist geschlossen”. Die Binärsignale sind aber wortweise organisiert; es kann also nicht ein einzelnes Binärsignal vom Rechner abgefragt werden, sondern nur jeweils eine Gruppe, die durch die Wortlänge des Rechners gegeben ist (daher die Bezeichnung ”Digital-”). Die Isolierung eines einzelnen Bits erfolgt enschließend im Rechner durch das Arbeiten mit Masken oder, wenn der Befehlsvorrat des Rechners dies gestattet, durch Befehle, die das Ansprechen eines einzelnen Bits in einem Wort ermöglichen. Entsprechendes gilt für die Digitalausgänge: hier müssen die Kommandos (Stellbefehle), also Bits, im Rechner zu Wörtern montiert werden, die dann die Ausgänge ausgeben werden.

Obwohl ein Bit das Informations-Atom ist, aus dem sich alle anderen Darstellungen von Daten zusammensetzen, ist es bereits auch ein Datentyp. Auf Bits mit dem Wertebereich $[0, 1]$ sind nämlich folgende Operationen erklärt:

- Setzen (Zuweisen einer 1)

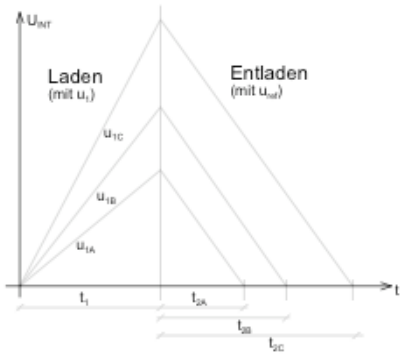


Abbildung 4.32: Lade- und Entladevorgänge beim Dual-Slope Wandler für verschiedene Eingangsspannungen.

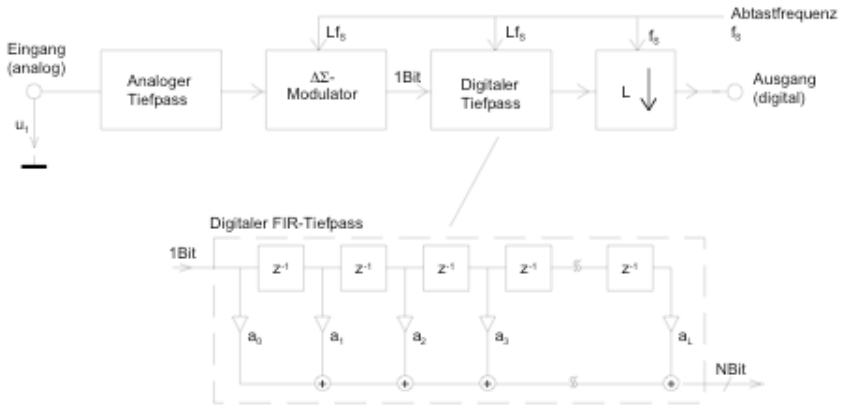


Abbildung 4.33: Blockschaltbild eines ΔΣ - Wandlers.

- Löschen (Zuweisen einer 0)
- Teset (Abfrage des Wertes: 0 oder 1 ?)

Wie erwähnt haben diese auf Bits und Binärwörter erklärten Operationen in der Automatisierungstechnik eine erhebliche Bedeutung.

- **Flags** Einmal wird insbesondere in verteilten Systemen, aber auch innerhalb eines Rechners mit sogenannten Flags (Flag: Flagge, Fahne) gearbeitet. Flags sind 1-Bit-Signale, die Auskunft über den jeweiligen Bearbeitungszustand eines Gerätes, einer Baugruppe oder eines Übertragungskanals geben wie "Ist die Analog-Digital-Wandlung abgeschlossen?" oder "Ist der Drucker gerade aktiv?". Sie werden üblicherweise in einem Flipflop kurzzeitig gespeichert.
- **Bits als Informationsträger in einem Wort** In Prozessrechnern sind die Eingänge für Binärsignale, die Meldungen über den aktuellen Prozesszustand geben, zwar wortweise als sogenannte Digitaleingänge organisiert, Informationsträger ist aber der Wert eines Bits. Um diese Meldungen auswerten zu können, muss das Bit also erst in einem Binärwort selektiert werden (es muss auf das entsprechende Bit zugegriffen werden). Dies kann auf

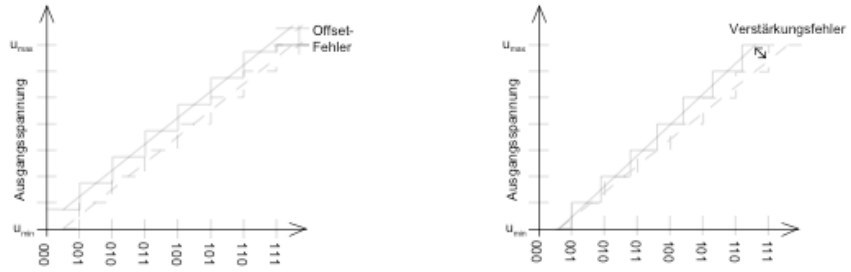


Abbildung 4.34: Einfluss der Offset- und Verstärkungsfehler beim DA-Wandler

- **Monotonie** Bei gleichmässig steigender Eingangsspannung wird eine gleichmässig steigende Ausgangsspannung in Quantisierungsschritten erwartet. Bei Monotoniefehler treten gewisse Ausgangsspannungen nicht auf.

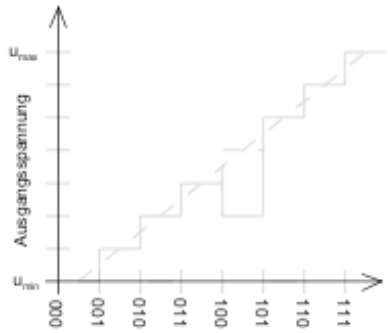


Abbildung 4.35: Monotoniefehler beim DA-Wandler. Sie äussern sich in "Missing Codes".

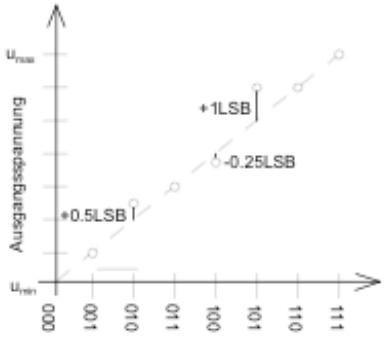


Abbildung 4.36: Integrale Nichtlinearität als Abweichung zwischen tatsächlichem und idealen Wert.

eine der folgenden Arten geschehen, die auf einander aufbauen:

- Auswahlalgorithmus: Ein möglicher Auswahlalgorithmus wäre das Abzählen der Bits in einem Wort vom ersten oder letzten Bit aus.

- Maskieren: Unter Maskieren versteht man die bitweise UND-Verknüpfung des betrachteten Binärwortes mit einer "Maske", bei der alle Bits mit Ausnahme des auszuwählenden den Wert 0 haben. Die Auswahl des 4. Bits aus einem 8-Bit-Wort vollzieht sich beispielsweise über:

```
8-Bit-Wort:          0010 1101
Maske:               0000 1000
                     - - - - - - - -
UND-Verknüpfung:     0000 1000
```

Wäre das 4. Bit eine 0 gwesen, stände in dem Ergebnis dort auch eine 0.

– **Bit-Selection als Sprachkonstrukt:** Die Selektion eines Bits geschieht in den höheren Programmiersprachen, die speziell für die Belange der Prozessinformatik entwickelt wurden (z.B. PEARL), dadurch, dass die Bitanwahl im Sprachvorrat der entsprechenden Sprache bereits als ein Befehl berücksichtigt wurde.

4.6 Feldbussysteme

4.6.1 Eigenschaften und Einsatzgebiete

Eigenschaften

Feldbussystene sind zunächst Bussysteme wie sie auch für lokale Rechnernetze (LAN) eingesetzt werden. Die dort gegebenen allgemeinen Ausführungen über Protokolle und Zugriffsverfahrengelten somit auch für Feldbusse. Bei Feldbussen steht der Begriff "Bus" allerdings auch für Ring-, Stern- und Baum-Topologien, ist also nicht auf einen linearen Leitungsschrank begrenzt, an den Datenverarbeitungssysteme oder andere Teilnehmer angeschlossen werden . Konsequenter wäre es, anstelle von Feldbussen von "Feld-LANs" zu sprechen, die Unternetze in einem Automationssystem sind.

Darüber hinaus gelten die folgenden allgemeinen Aussagen, wobei die jeweils wichtigen Zahlenwerte noch von dem betrachteten Bussystem abhängen:

- die Nachrichtenübertragung erfolgt bitseriell,
- als Übertragungsmedien kommen vor allem Twisted-Pair (geschirmte oder ungeschirmte verdrillte Zweidrahtleitungen), gelegentlich auch Koaxkabel zum Einsatz,
- bei einigen Systemen sind Längen bis zu 10 Km möglich (gegebenenfalls mit Repeatern),
- anschließbar sind teilweise bis zu 1000 Stationen,
- die Paketlängen sind unterschiedlich lang, können aber auch nur 2 - 16 Byte umfassen,
- es werden selbsttaktende Codes (meist der Manchester-Code) herangezogen,
- die im Feld immer notwendige Potentialtrennung erfolgt über Optokoppler oder mittels Übertragern (induktive Anschaltung).

Darüber hinaus sind folgende Punkte im Einzelfall zu betrachten:

- die Eigensicherheit eines Busses im Bergbau oder in explosionsgefährdenden Anlagen der chemischen Industrie sowie (damit verbunden)
- die Stromversorgung der an den Bus angeschlossenen Geräte.

Einsatzgebiete

Die wichtigsten Einsatzgebiete von Feldbussen sind

- Produktionsprozesse in der Fertigungstechnik,

- Produktionsprozesse in der Verfahrenstechnik,
- die Automatisierung der technischen Einrichtungen von größeren Gebäuden (Banken, Versicherungen, Verwaltungen) sowie
- die Steuerung und Überwachung der technischen Aggregate in Kraftfahrzeugen.

Die sehr unterschiedlichen Einsatzgebiete und die divergierenden Interessen von Lieferfirmen der Automatisierungstechnik führten dazu, dass Anfang der 90er Jahre des 20. Jahrhunderts mehr als 40 verschiedene Varianten von Feldbussen existierten. Für die von den Lieferfirmen angebotenen Varianten existieren bereits Anschlussgeräte (Interfaces), während für die im langjährigen nationalen und internationalen Normungsprozess befindlichen Varianten solche Koppeleinheiten nur zögernd auf den Markt kamen.

Heute hat sich die Typenvielfalt etwas gelichtet, so dass die folgenden Ausführungen sich auf einige Beispiele beschränken können, hinter denen große Stückzahlen in den Anwendungen stehen (lt. CAN-Nutzerorganisation z.B. 1997 21 Millionen)

4.6.2 Schnittstelle von Anwendungen und Kommunikationssystem

Problemstellung

Mit Hilfe von Feldbussen wird anlagennah ein "Kommunikationssystem" bereitgestellt, dass, wie erwähnt, den Character eines LAN oder eines Unternetzes in einem Automatisierungssystem. Dieses Kommunikationssystem ist natürlich nur ein Hilfsmittel, um die eigentliche Aufgabe: die Automatisierung eines Prozesses, zu lösen, die vielfach hoch komplex ist.

Für diese Aufgabe wird beim heutigen Stand der Technik eine Menge von Programm-Modellen herangezogen, die an einer Stelle oder verteilt auf der Leitebene des Bedieners oder Beobachters einzuordnen sind. Ein solcher Program-Modul heißt in dem hier behandelten Zusammenhang "Anwendung (Application)".

Gegenstand der folgenden Ausführungen ist die Schnittstelle zwischen der jeweiligen Anwendung und dem Kommunikationssystem. (Der Begriff "Anwendung im Zusammenhang mit dem ISO/OSI-Refernzmodell hat seine Tücken. Er steht einmal für eine Anwendung in dem hier vorgestellten Sinn, zum anderen aber auch für die Gesamtheit der Dienste, die im Zusammenhang mit dieser Aufgabe auf den Schichten 5-7 erbracht werden.)

Schichtenmodell

Eigenschaften von Feldbussen können heute nicht mehr ohne Bezug auf das ISO/OSI-Referenzmodell mit sieben Schichten beschrieben werden.

- **Relevanz der Schichten** Bei Feldbussen müssen vom Prinzip her keine virtuellen Verbindungen (verbindungsorientierte oder verbindungslose) zwischen Kommunikationspartnern über ein Netz von Transitrechnern wie in einem WAN gehandhabt werden, da weder ein solches offenes Netz noch Transitrechner existieren. Probleme des Aus- und Abbaus einer virtuellen Verbindung oder des Routings für ein Datagramm werden in den Schichten 3-5 des Referenzmodells geregelt; diese können daher eigentlich bei Feldbussen entfallen.

  Dementsprechend ist die vielfach verwendete Aussage *Bei Feldbussen sind nur die Schichten*

1,2 und 7 des Referenzmodells relevant, die Schichten 3-6 sind leer, im wesentlichen gerechtfertigt (eine Ausnahme bildet nur LON, wo aufgrund der dort möglichen Netz-Topologien auch einige Merkmale der Schicht 3 standardisiert werden mussten). Allerdings erfahren die Schichten 2 und 7 gewisse Funktionserweiterungen, um einige der Aufgaben der fehlenden Schichten wahrzunehmen. Gewissermaßen durch die Hintertür werden dort virtuelle Verbindungen zwischen den Kommunikationspartnern eingeführt. Diese Erweiterungen werden bei den besprochenen Bussystemen erläutert.

Will man Feldbusse verschiedener Hersteller, die nach firmeneigenen Standards gestaltet sind, miteinander koppeln, benötigt man teure Gateways.

- **Schicht 8** Eigentlich ist die Bezeichnung "Anwendungsschicht" für die Schicht 7 des Referenzmodells nicht gerechtfertigt, da dort nur Fragen des Zugangs zum Kommunikationssystem in Normen geregelt werden. Die eigentliche Anwendung müsste in einer Schicht "Schicht 8" angeordnet werden. Wegen der praktisch unübersehbaren Fülle der Aufgaben, die mit einem WAN oder einem LAN als Kommunikationsmedium gelöst werden sollen, ist eine Standardisierung einer solchen Schicht dort nicht möglich.

  Beim Heranziehen des Referenzmodells für die speziellen "Themenkreise der Automatisierungstechnik" und insbesondere der Feldbussysteme ändert sich die Betrachtungsweise aber erheblich: die *Anzahl der Aufgabenkomplexe ist überschaubar begrenzt!* Man kann damit daran denken, nicht nur die *Syntax* von Botschaften, die im Rahmen einer bestimmten Anwendung benötigt werden, zu standardisieren, sondern auch deren *Semantik*, also ihre Bedeutung.

  In Analogie zu WANs und kommerziellen LANs wird man zunächst geneigt sein, diesen Aufgabenkreis einer *Schicht 8* eines erweiterten Referenzmodells zuordnen. Es zeigt sich aber bei den Anwendungen im Feldbussystemen, dass Definition und Prüfung von Syntax und Semantik nicht eindeutig voneinander getrennt werden können.

  Definition und Prüfung der *Syntax von Botschaften* vollzieht sich bei WANs und LANs üblicherweise auf der Ebene 6 des Referenzmodells. Diese existiert aber bei Feldbussystemen nicht. Die entsprechenden Aufgaben müssen also entweder der Schicht 7 zugewiesen oder in der eigentlichen Anwendung vorgenommen werden. Das selbe gilt aber auch für die *Semantik der Botschaften*. Beide Aufgabenkomplexe vollziehen sich also jeweils auf den selben Schichten 7 oder 8. Zu beachten ist aber, dass die Schicht 7 den Intentionen des Referenzmodells entsprechend Gegenstand einer herstellerunabhängigen Normung ist, die Schicht 8 dies aber nur in einem gewissen Umfang sein kann.

- **Interoperabilität** Ziel jeder Standardisierung im Bereich der Feldbusse ist, dass der Betreiber des Automationssystems jederzeit die Geräte eines Lieferanten ohne Zeitaufwand durch die eines anderen ersetzen kann – sei es während der Planung, sei es während des späteren Betriebs. Dieses Ziel wird als Interoperabilität bezeichnet.

  Um dieses Ziel zu erreichen, müssen zunächst von jedem Lieferanten alle Vereinbarungen eingehalten werden, die den Schichten 1 und 2 des Referenzmodells zugeordnet sind. Diese betreffen den mechanischen Anschluss der Geräte, Signalniveau und Signalbedeutung der Schnittstellen, den verwendeten Code sowie das Format der Telegramme (Datenrahmen). Da diese in einer Feldbus-Norm eindeutig definiert sind, kann der betreiber davon ausgehen, dass jeder Lieferant sich darn hält.

  Darüber hinaus möchte der Betreiber aber auch keine Änderungen in seiner Anwendersoftware (seiner *Anwendung*) vornehmen müssen, da dies immer mit erhelblichem Zeit- und Kostenaufwand für ihn verbunden ist. Anwendungen sind immer in einer bestimmten Programmiersprache formuliert und laufen unter einem bestimmtem Betriebssystem.

Die Schicht 7 der jeweiligen Feldbus-Norm (wenn diese Schicht überhaupt genormt ist) stellt ihm zwar bestimmte Dienste (Services) zur Verfügung; bei PROFIBUS FMS sind dies etwa 40. Die Norm beschreibt aber nur die Aufgaben der Dienste und deren Parameter , nicht aber, wie sie realisiert werden. Der Zugriff der Anwendung auf diese Dienste kann also von der Programmierung her sehr unterschiedlich gestaltet sein, ist aber keineswegs Bestandteil der Feldbus-Norm und damit von der jeweiligen Implementierung abhängig

- **Application Layer Interface (ALI)** Man versucht, dieses Dilemma durch *Einführung eines Application Layer Interface (ALI)* zu überwinden. das zwischen der eigentlichen Anwendung und der Schicht 7 der Feldbus-Norm einzuordnen wäre. Es existiert aber keine Norm für diese Zwischenschicht und damit gilt, dass das ALI auch bei einem genormten Feldbussystem wie z.B. dem PROFIBUS, implementationsabhängig ist.

  Letztlich ist also eine Standardisierung, die

  – alle Festlegungen der jeweiligen Feldbus-Norm

  – mit einer einheitlichen Schnittstelle

  – zu einer speziellen Anwendung berücksichtigt,

  gegenwärtig nicht verfügbar. Eine Norm muss alle denkbaren Einsatzgebiete des betreffenden Busses beachten, ist dafür aber für einen speziellen Einsatzfall zu allgemein.

  In diesem Zusammenhang wurden die Begriffe *Kernprotokoll* und *Protokollspezialisierung* eingeführt [19]:

  – ein Kernprotokoll stellt allgemein gehaltene Kommunikationsobjekte und -dienste zur Verfügung.

  – während durch Protokollspezialisierung "für bestimmte Einsatzbereiche oder auch bestimmte Gruppen von Geräten ein problemloses Zusammenspiel der Teilsysteme sichergestellt sowie eine angemessene, anwendungsspezifische Semantik zur Verfügung gestellt werdenßollen.

  Ein Beispiel eines Kernprotokolls ist die Schicht 7 von PROFIBUS FMS, während Protokollspezialisierungen bei PROFIBUS "Profile", bei MMS "Companion Standards" heißen.

- **Profile** Bei einem Profil beschränkt man sich auf einen klar überschaubaren Aufgabenkomplex der Automatisierungstechnik. Man hat dabei zu unterscheiden zwischen:

  – Kommunikationsprofilen

  – Geräteprofilen

  – Branchenprofilen

  In einem Kommunikationsprofil werden die Schichten 1 bis 7 und eine Untermenge der Kommunikationsobjekte und -dienste definiert. Ein Geräteprofil definiert für ein bestimmtes Gerät oder eine Klasse von Geräten deren Darstellung in der Schicht 7, also ihr Bild für einen Kommunikationspartner nach außen. In einem Branchenprofil werden mehrere dort wichtigen Geräteprofile zusammengefasst und deren Zusammenwirken festgelegt.

  Als Beispiele seien genannt:

1. die Überwachung und Steuerung von analogen und digitalen Ein- und Ausgängen eines Rechners der B&B-Ebene,

2. die Steuerung von Robotern,

3. die Steuerung von elektrischen Antrieben, die in ihrer Drehzahl kontinuierlich verstellbar sind,

4. die Automation von Gebäuden.

Von diesen Beispielen aus der PROFIBUS-Welt sind (1.) eine Kommunikationsprofil, (2.) und (3.) sind Geräteprofile und (4.) ist ein Branchenprofil.

Wesentlich ist, dass für ein solches Profil sowohl die Anwendung als auch die Schicht 7 bis zum Zugriff auf die Schicht 2 standardisiert werden. Da die Schichten 2 und 1 üblicherweise in einem speziellen Chip integriert sind, werden in einem Profil also von der Semantik der speziellen Automatisierungsaufgabe bis zur Signalübertragung in einem gegebenen Bussystem alle Schritte über alle Schichten des Referenzmodells hinweg standardisiert. Damit sind auch Details der späteren Implementierung festgelegt.

4.6.3 Die PROFIBUS-Familie

Entstehung und Bedeutung

Mit Unterstützung durch das BMFT (Bundesministerium für Forschung und Technologie) wurde in Deutschland seit etwa 1986 der PROFIBUS (Process Field Bus) entwickelt. Dieser ist ein offener Feldbusstandard, der die Kommunikation von Geräten unterschiedlicher Hersteller gestatten sollte, ohne dass Schnittstellen angepasst werden müssen. (Dass dies nur mit gewissen Einschränkungen möglich ist, wurde vorstehend erläutert)

Der PROFIBUS wird von der (in der Zwischenzeit internationalen) PROFIBUS-Nutzerorganisation (PNO) weiterentwickelt und vermarktet, in der weltweit mehrere hundert Liefer- und Anwenderfirmen sowie Hochschulinstitute der Automatisierungstechnik vertreten sind. Als Ergebnis der Weiterentwicklung liegen neben der ursprünglichen Version PROFIBUS FMS (FMS: Field Message Specification) auch die Versionen PROFIBUS DP (DP: Dezentrale Peripherie) und PROFIBUS PA (PA: Prozess-Automatisierung) vor.

Die Merkmale dieses Feldbusses sind in DIN 19245, Teil 1 bis 4, festgelegt, aber auch in der europäischen Norm EN 50 170 (dort zusammen mit zwei anderen Feldbussnormen, nämlich dem P-NET, der in Dänemark und dem WordFIP, der in Frankreich entwickelt wurde).

Nach Angabe der PROFIBUS-Nutzerorganisation waren Anfang 1997 bereits mehr als 1 Mio. Geräte in mehr als 100 000 Anwendungen weltweit nach den PROFIBUS-Spezifikationen installiert.

Famillienmitglieder

- **PROFIBUS FMS** ist die älteste und universellste, aber auch die aufwendigste Version dieses Feldbussystems. Sie ist gedacht für den objektorientierten, universellen Datenaustausch im Zellbereich. Fertigungszellen sind ein Merkmal der Fertigungstechnik. Diese stand bei Beginn der Normungsarbeiten am PROFIBUS im Vordergrund des Interesses.

Dies zeigt sich auch daran, dass die Version PA, die den Belangen der verfahrenstechnischen Prozesse Rechnung trägt, die letzte der drei bisher entwickelten Versionen war, die erst mit einer wesentlichen Verzögerung in Angriff genommen wurde.

- **PROFIBUS DP** Diese Version ist auf schnelle Übertragung und auf niedrige Kosten der Bus-Interfaces optimiert. Sie ist zugeschnitten auf die Datenübertragung in den beiden unteren Leitebenen, insbesondere auf die Kommunikation zwischen B&B-Ebene (Ebene 2) mit einfachen dezentralen Peripheriegeräten in der Feldebene (Ebene 1). Sie hat daher den Character eines Sensor/Aktor-Busses.

  Profil-Bilder

- **Profibus PA** Dies ist die eigensichere Version für den Bereich der verfahrenstechnischen Industrien, in denen aufgrund der dort ablaufenden Prozesse die Gefahr von Explosionen besteht, wenn irgendwo ein Funke entsteht. Eigensicher heißt, dass die für die Signalübertragung aufgewendete Energie so klein ist, dass sie nicht zum Zünden eines explosiven Gasgemisches ausreicht.

  PROFIBUS PA erlaubt somit die Anbindung von Sensoren und Aktoren auch in explosionsgefährdeten Bereichen an eine gemeinsame Busleitung. Geräte können dort auch während des laufenden Betriebs vom Feldbus abgeklemmt oder an diesen angeschlossen werden. Weiter ermöglicht PROFIBUS PA nicht nur den Datenaustausch zwischen den angeschlossenen Geräten, sondern auch deren Energieversorgung in 2-Draht-Technik.

  Zugrunde liegt der internationale Standard IEC 1158-2. Die Übertragungsrate ist nach dieser Norm mit 31,25 kBit/s recht niedrig. Eine höhere Rate wird bei den langsamen Prozessen der Verfahrenstechnik aber üblicherweise nicht gebraucht, ist aber nach der Norm zulässig. Dafür ist der Betrieb mit ungeschirmten Leitungen auch bei großen Leitungslängen (max. 1900m) möglich.

  PROFIBUS PA - Geräte können über einen Segmentkoppler (der aus der Sicht von LANs den Character einer Bridge hat) mit einem PROFIBUS DP - Netz verbunden werden. Hierdurch kann ein eigensicherer Bereich an einen Bereich angebunden wwerden, der normalen Sicherheitskriterien genügt.

  Wie für Feldbusse typisch, sind bei allen drei Versionen von PROFIBUS die Schichten 3 bis 6 des ISO/OSI-Referenzmodell leer. Die Schichten 1 und 2 sind bei allen Versionen, die Schicht 7 dagegen de jure nur bei FMS durch Standards ausgefüllt (de facto ist bei DP mit DDLM diese Schicht auch spezifiziert). Aller Versionen verfügen über die oben erwähnte ”8. Schicht”, in der Anwendungsprofile festgelegt sind.

Hybrides Zugriffsverfahren

PROFIBUS verwendet ein hybrides Zugriffsverfahren. Die angeschlossenen Stationen gehöeren zwei Klassen an:

- Master können ohne externe Aufforderung senden, wenn sie im Besitz der Sendeberechtigung sind. Die Sendeberechtigung wird zwischen den Mastern nach dem Prinzip des logischen Token Rings weitergegeben; man spricht von einem ”Flying Master”. Master sind meist PCs oder SPSen auf der Zellebene. Die Kommunikation kann sich zwischen dem gerade aktiven Master und den ihm zugeordneten Slaves vollziehen, aber auch zwischen den Mastern.

- Slaves dagegen dürfen nur nach Aufforderung durch den Master ein Quittungssignal oder Nachricht senden. Slaves sind meist Sensoren und Aktoren im Feld. Sie benötigen nur eine kleine Untermenge des Busprotokolls, wodurch die Ankopplung an den Bus kostengünstig wird.

Schicht 1

Die Datenübertragung auf der Schicht 1 erfolgt bei den Versionen FMS und DP entweder nach der Norm RS-485 oder mit Lichtwellenleitern (LWL)

- RS-485 wird am häufigsten eingesetzt, da sie eine hohe Übertragungsgeschwindigkeit (einmalig wählbar im Bereich von 9,6 kBit/s bis 12 Mbit/s) mit einer kostengünstigen Installation verbindet. Verwendet wird ein verdrilltes Leiterpaar aus Kupfer mit Abschirmung. Ein solcher Bus wird auch H2-Bus genannt.

- Für Umgebungen mit hohen Störpegel oder für längere Übertragungsstrecken werden LWL herangezogen.

Bei der eigensicheren Version PA nach der Norm 1158-2 (H1-Bus genannt) kommt ein bitsynchrones Übertragungsverfahren ohne Gleichstrom-Komponente zum Einsatz. Diese ist allerdings relativ langsam: die Übertragungsrate ist 31,25 kBit/s.

Schicht 2

- **Format der Telegramme** Die Telegramme (Datenpakete) haben beim PROFIBUS eine Hamming-Distanz $HD = 4$. Die Hamming-Distanz beschreibt bei gleicher Wortlänge die Anzahl der Bits, in der zwei binär codierte Wörter nicht übereinstimmen. In der Codierungstheorie wird gezeigt, dass die Anzahl e der bei einer Datenübertragung verursachten, aber erkennbarer Fehler sich aus

  $$e = HD - 1 \tag{4.11}$$

  berechnet. Bei $HD = 4$ sind als 3 Übertragungsfehler erkennbar (aber nicht unbedingt korrigierbar).

- **Einführung von Teilschichten** Wie auch im Zusammenhang mit Ethernet, hat sich die Unterteilung der Schicht 2 in zwei Unterschichten LLC (Logical Link Control) und MAC (Medium Access Control) als nützlich erwiesen. Man trennt hierdurch die gesicherte Übertragung (LLC) von allen Fragen, die mit dem Zugriff auf das eigentliche Übertragungsmedium verbunden sind (MAC). Diese Unterteilung wurde auch bei PROFIBUS vorgenommen. Die Unterschicht LLC heißt hier FDL (Field Data Link).

- **Unterschicht MAC** Die Unterschicht MAC hat neben der eben beschriebenen Aufgabe, den konfliktfreien Zugriff der Master auf den Bus zu regeln, bei PROFIBUS eine besondere Bedeutung in der Hochlaufphase des Systems, wo der logische Token-Ring aufgebaut werden muss bzw. beim Erkennen, ob Token verloren gingen oder doppelte Token kreisen und ob Adressen mehrfach belegt sind.

- **Unterschicht FDL** Die Unterschicht FLD bietet übergeordneten Schichten Dienstzugangspunkte SAP (Service Access Points) für folgende Formen der Datenübertragung mit gesicherten Protokollen an:

  – SDN: Send Data with no Acknowledge (Multi- oder Broadcasting)

- SDA: Send Data with Acknowledge
- RDR: Request Data with Reply
- SDR: Send and Request Data (Kombination aus SDA und RDR)

Wegen der oben beschriebenen Bedeutung des Pollings in heutigen Automationssystemen stehen außerdem folgende Übertragungsdienste zur Verfügung:

- CRDR: Cyclic Request Data with Reply
- CSDR: Cyclic Send and Request Data with Reply

Die Versionen DP und PA bieten nur die Dienste SDR und SDN an.

Schicht 7

Die Schicht 7 ist nominell nur bei der Version FMS besetzt. Auch sie ist dort in zwei Unterschichten unterteilt: LLI und FMS. (FMS hat hier also zwei Bedeutungen)

- **LLI: Lower Layer Interface** Diese Unterschicht hat primär in rudimentärer Form die Aufgaben zu übernehmen, die sonst in den Schichten 3 bis 6 des Referenzmodells wahrgenommen werden.

  PROFIBUS FMS gestattet sowohl die verbindungsorientierte als auch die verbindungslose Kommunikation

  - erstere zwischen zwei klar definierten Partnern des Feldbussystems (Slaves müssen einem Master eindeutig zugeordnet werden)
  - letztere beim Multi- oder Broadcasting

  Der Auf- und Abbau einer virtuellen Verbindung im erstgenannten Fall vollzieht sich für die beiden Kommunikationspartner in WANs über das Netz auf der Transportschicht (Schicht 4), der zwischen zwei Transitrechnern auf der Netzwerkschicht (Ebene 3). Da in Feldbussystemen keine Transitrechner vorhanden sind, aber trotzdem virtuelle Verbindungen definiert werden sollen, muss diese Aufgabe von der Unterschicht LLI wahrgenommen werden.

  Außerdem erfolgt in dieser Schicht die Transformation (Übersetzung) der von der Unterschicht FDL bereitgestellten Dienste in die Syntax der Dienste der Unterschicht FMS.

- **FMS: Fieldbus Message Specification** Die FMS-Dienste sind eine Untermenge der Funktionen, die bei MAP (Manufacturing Message Protocol) in der Schicht 7 des Referenzmodells als MMS (Manufacturing Message Specification) nach ISO 9506 definiert sind.

- **DDLM: Direct Data Link Mapper** Diese Dienste-Schicht erfüllt bei der Version DP de facto die Aufgabe einer Anwendungsschicht, auch wenn sie in DIN 19.245, Teil 3, so nicht bezeichnet wird. Dies hat wohl seinen Grund darin, dass die zu übertragende Nutznachricht auf der Schicht 7 noch mit einem Header und einem Prüffeld versehen wird. Bei PROFIBUS DP, das praktisch den Character eines Sensor/Aktor-Busses hat, verzichtet man im Interesse der Übertragungseffizienz auf diesen Overhead. Hierdurch werden die Datenpakete so kurz, wie es das PROFIBUS-Protokoll überhaupt zulässt.

Schicht 8 – Profile

Nach dem Stand von 1997 existieren die im folgenden genannten Profile. Für die Kennzeichnung ihres Types werden folgende Bezeichnungen verwendet:

- K: Kommunikationsprofil
- G: Geräteprofil
- B: Branchenprofil
- Version FMS Jeweils ein Profil
  - für die Kommunikation zwischen SPS-Steuerungen (K)
  - für die Automation von Gebäuden (B) und
  - für Niederspannungsshcaltgeräte (G)
- Version DP Jeweils ein Profil
  - für die Steuerung von Robotern (G)
  - für den Anschluss von Winkel- und Linear-Encodern (G)
  - für drehzahlveränderliche Antriebe (G) und
  - für Bedienen und Beobachten (K)
- Version PA Jeweils ein Profil
  - für digitale Ein- und Ausgänge (G)
  - für analoge Ein- und Ausgänge (G)
  - für Ventile (G)
  - für Stellantriebe (G)
  - für Druck-, Füllstands-, Temperatur- und Durchfluss-Sensoren (G)

4.6.4 CAN-Bus

Entstehung und Bedeutung

CAN steht für Controller Area Network. Dieses Bussystem wurde von der Robert Bosch GmbH seit Ende der 80er Jahre des 20. Jahrhunderts zunächst für den Einsatz in Kraftfahrzeugen entwickelt. Es sollte die bis dahin üblichen Kabelbäume ersetzen, die insgesamt eine Länge von 2000 , und ein Gewicht von mehr als 100 kg erreichen können. Dies resultiert aus der in den letzten zwei Jahrzenten stark gestiegenen Anzahl elektronischer Steuereungen in den Kraftfahrzeugen, die teilweise extrem harten Realzeitbedingungen genügen müssen (wie Motormanagement, automatisches Getriebe, Bremsen, ABS, Stabilisierung und Arbag). Andere Steuerungen wie

Fensterheber, Türschließanlage, Klimatisierung, Beleuchtung usw. sind weniger zeitkritisch, erfordern somit auch nur eine geringe Bandbreite des Übertragungsmediums, sind aber sehr kostenintensiv. Seit etwa 1994 hat der CAN-Bus konkurrierende Bussysteme im Kraftfahrzeugbau praktisch verdrängt. Aktuell drängt in diesen Markt das Bus-System "FlexRay".

Wegen der hohen Stückzahl im Automobilbau sind die Kosten für den Anschluss von Geräten an den CAN-Bus so niedrig wie bei keinem anderen Feldbussystem. Dies führte dazu, dass der CAN-Bus auch für industrielle Automatisierungsprojekte zunehmend interessant wurde, zunächst insbesondere in der Textilindustrie, zwischenzeitlich aber auch in zahlreichen anderen Branchen.

Der CAN-Bus dient der Vernetzung einfacher Sensoren und Aktoren mit der überlagerten Steuerung, insbesondere, um Informationen für die Parametrierungvon intelligenten Geräten zu übertragen. Es sind die Schichten 1,2 und 7 standardisiert.

Zugriffsverfahren CSMA/CA

CSMA/CA steht für Carrier Sense Multiple Access/Collision Avoidance. Dieses Zugriffsverfahren ist durch zwei Merkmale gekennzeichnet:

1. Der erste Teil dieser Bezeichnung bezieht sich wie beim Ethernet auf den gleichberechtigten Zugriff aller Stationen (DVSen) auf das Übertragungsmedium, während "Collision Avoidance" bei einer Kollision zweier gleichzeitig gesendeter Botschaften beide nicht erst wie bei CSMA/CD in einen Wartezustand versetzt, sondern von vornherein beim Erkennen der Kollision der wichtigeren der beiden Botschaften Priorität der Übertragung einräumt.

2. Das Zugriffsverfahren auf das Übertragungsmedium unterscheidet sich beim CAN-Bus von allen anderen bisher behandelten durch die Semantik des zu übermittelnden Datenpakets: die in den Steuerfeldern eines Telegramms enthaltene Nachricht bezieht sich nicht auf den jeweiligen Adressaten, sondern auf die Bedeutung der Botschaft im Vergleich mit allen anderen im System vorgesehenen Botschaften. – Implizit ist in der Prorität der Botschaft natürlich auch die Bedeutung des Adressaten enthalten.

Das sich eine wichtigere Botschaft gegenüber allen anderen durchsetzt, die nach Beendigung einer laufenden Übertragung gleichzeitig um das Übertragungsmedium konkurrieren, wird durch rein elektrische Mittel erreicht: eine binäre "0" erhält bei gleichzeitigem Auftreten ummer Priorität vor einer binären "1". Dies kann beispielsweise durch eine Open-Collector-Schaltung von Transistoren erreicht werden, die ein logisches UND realisiert.

Das nachfolgende Bild zeigt an einem Beispiel die eingeschlagene Strategie. (T steht für Transmitter, R für Receiver). Beide Stationen verfolgen mit ihren Emfpängern R1 und R2 die Signale auf dem Bus. Die Überprüfung, ob das von einer Station gesendete und empfangene Signal übereinstimmen, erfolgt jeweils in der Mitte der Bitzeit T_B um den Auswirkungen der Signallaufzeit T_L zu begegnen. Nach Abschluss einer laufenden Übertragung senden die beiden sendewilligen Stationen 1 und 2 zunächst ein Startbit, das einer dominanten "0" entspricht. Sie müssen hierfür natüerlich streng synchron arbeiten. Die Adresse der Botschaft von Station 1 beginnt mit der Folge 1011. . . , die der Station 2 mit 1001. . . . Beim dritten Bit ist die 0 von Station 2 dominant gegenüber der rezessiven 1 von Station 1. Station 2 setzt sich also durch und sendet weiter. Station 1 erkennt mittels R1, dass die von ihr gesendete 1 in eine 0 abgeändert wurde, und bricht ihre Übertragung ab (am Sender T1 steht weiterhin die rezessive 1 an).

Botschaften mit einer niedrigen Adresse verdrängen somit solche mit einer höheren. Die Adresse hat also den Character einer Priorität der Botschaft.

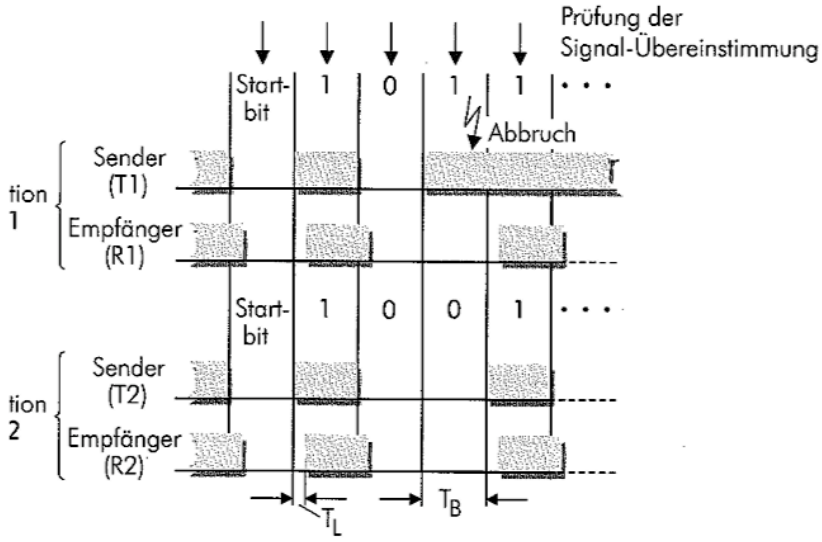


Abbildung 4.37: Zum CSMA/CA - Zugriffsverfahren

Übertragungsrate

Damit das Verfahren funktioniert, muss die Signallaufzeit T_L kelin gegen die Bitzeit T_B sein. Mit der Ausbreitungsgeschwindigkeit v, der Laufstrecke l und der Übertragungsrate r gilt

$$T_L = \frac{l}{v} << T_B = \frac{1}{r} \tag{4.12}$$

somit also auch

$$l \cdot r << v \tag{4.13}$$

Das Produkt aus Übertragungsrate und Leitungslänge hat einen festen Wert. Beide sind an einander gekoppelt, jeweils in Abhängigkeit vom Wert des anderen begrenzt. Zusammen mit der Definition von Profilen werden beispielsweise folgende Wertepaare empfohlen: eine Übertragungsrate von r = 1 Mbit/s bei einer maximalen Leitungslänge l von 25m und r = 10 kbit/s bei einer Länge von 5000 m.

Realzeitfähigkeit und Sicherheit

- **Realzeitfähigkeit** Mit dem Buszugriffsverfahren CSMA/CA und der Priorisierung der Botschaften ist der CAN-Bus realzeitfähig. So wie beim Multitasking in der Prozessrechentechnik sich die jeweils wichtigste Task gegenüber allen anderen durchsetzt, wird hier die jeweils wichtigste Botschaft unmittelbar übertragen. Im Gegensatz zum Preemtive Scheduling wird beim CAN-Bus eine laufende Botschaft allerdings nicht unterbrochen.

- **Sicherheit** Der CAN-Bus verfügt über eine sehr hohe Übertragungssicherheit. Das CAN-Telegramm hat zwar nur maximal 8 Nutzdaten-Bytes, enthält aber mehrere Möglichkeiten, einen Fehler bei der Datenübertragung zu erkennen. Neben einer 15 Bit langen CRC-Sequenz gibt es einen Acknowledge- und einen Frame-Check, wobei die letzten beiden an das Prinzip der dominanten und rezessiven Bits gebunden sind, bei anderen Feldbussystemen also nicht zum Einsatz kommen können. Hierdurch kann auf Quittungstelegramme verzichtet werden: der CAN-Bus arbeitet dementsprechend verbindungslos auf der Basis des Client-Server-Prinzips.

Schicht 7

Beim CAN-Bus waren anfangs nur die Schichten 1 und 2 des Referenzmodells genormt, später haben mehrere Anwendergruppen Schicht-7-Protokolle definiert. Von diesen wird hier nur der "CAN Application Layer" (CAL) genannt. Diser umfasst einerseits "CMS", andererseits eine Gruppe von Anwendungen für das Management eines CAN-Netzwerks mit den Unter-Paketen "NMT" (Network Manager), "DBT" (Distributor) umd "LMT" (Layer Manager)

CMS (CANbased Message Specifications) ist wie beim FMS bei PROFIBUS eine Untermenge der "Manufacturing Message Specification" (MMS). Es werden wie beim FMS Objekte definiert (zum Beispiel CMS-Objekte vom Typ Variable, Domain oder Event mit den zugeordneten Attributen).

Wesentlich ist der Objekttyp Event: hier kann ein Server, wenn er ein solches Ereignis erkennt, ohne Aufforderung durch einen Client sofort (im Rahmen der Möglichkeiten des CAN-Busses) eine Meldung an den Client abgeben. Ist der Client ein Prozessrechner oder ein Industrie-PC mit Einsteckkarte, löst er praktisch dort einen Interrupt aus. Auch dies ist ein Beitrag zur Realzeitfähigkeit von CAN

Profile

Auch beim CAN-Bus existieren mehrere Profile, die teilweise von unterschiedlichen Nutzergruppen definiert wurden. Als Beispiel sei CANopen genannt.

4.6.5 LON

Entstehung und Bedeutung

LON ist das Acronym für Local Operating Network. Dieses Feldbussystem wurde relativ spät, nämlich erst 1991/92, von der US-amerikanischen Firma Echelon entwickelt. Es hat in der Zwischenzeit insbesondere in der Gebäudeautomation einen erheblichen Marktanteil errungen. Gebäudeautomation bezieht sich in ersteer Linie auf größere Objekte, bei denen die Heizung, Lüftung und Luftfeuchtigkeit gestaffelt nach Nutzungsgrad und momentaner Sonneneinstrahlung gesteuert und überwacht werden müssen.

Die Attraktivität von LON ist vor allem in der Tatsache begründet, dass hier von Anfang an ein durchdachtes Feldbussystem angeboten wurde, welches alle Komponenten umfasst, die ein Anwender für dessen Einsatz benötigt. Es sind dies:

- ein LON-spezifischer Mikrocontroller, Neuron genannt
- verschiedene Transeiver (Transmitter/Receiver, also Sende/Empfangs-Bausteine), die einen Anschluss der Neuronen an fast jedes Übertragungsmedium gestatten,
- das übertragungsprotokoll "LonTalk", dass vollständig von einem Neuron gehandhabt wird,
- eindeutig definierte Netwerkvariable, durch die eine komfortable Schnittstelle zwischen Anwendung und Kommunikationssystem bereitgestellt wird, sowie
- der "LonBuilder", ein in sich geschlossenes System für die Entwicklung und die Inbe-

triebnahme eines LON-Netzwerks.

Die Bedeutung eines solchen Gesamt-Pakets von Hardware- und Software-Angeboten kann nur der ermessen, der sich mit der Realisierung eines Feldbussystems einmal auseinander setzen musste. Natürlich lässt sich Echelon dieses Software-Paket auch bezahlen.

Die Programmierung der Neuronen erfolgt in einer LON-spezifischen Sprache, "Neuron-C", die einerseits eine Untermenge von ANSI-C darstellt, andererseits aber um die bei LON benötigten Sprachkonstrukte erweitert wurde.

Topologie

Da LON primär für die Automation von größeren Gebäuden geschaffen wurde, ergibt sich die Notwendigkeit, organisatorische Untersysteme zu schaffen, die der Etagenstruktur eines größeren Gebäudes Rechnung tragen. Dementsprechend existiert bei LON eine Baumstruktur des Automatisierungssystems, die sich an der räumlichen Struktur eines Gebäudes orientiert:

- ein "Domain" bezeichnet ein Gebäude
- ein "Subnetz" versorgt eine Etage
- eine "DVS" (Datenverarbeitungsstation) bearbeit klar definierte Einzelaufgaben innerhalb einer Etage, es kann sich also um einen Sensor, einen Aktor, eine autonome Regelung, eine dedizierte Steuerung oder um eine kleine SPS handeln.

Ein Beispiel eines LON-Systems ist im naächsten Bild veranschaulicht. Das Bild lässt klar erkennen, dass man über Grenzen von Subnetzen hinweg Gruppen von Stationen bilden kann. Diese Stationen können dann gemeinsam mit einem Multicasting-Telegramm angesprochen werden, und die Schicht 3 sorgt dafür, dass jede Station das telegramm erhält. Eine Kommunikation

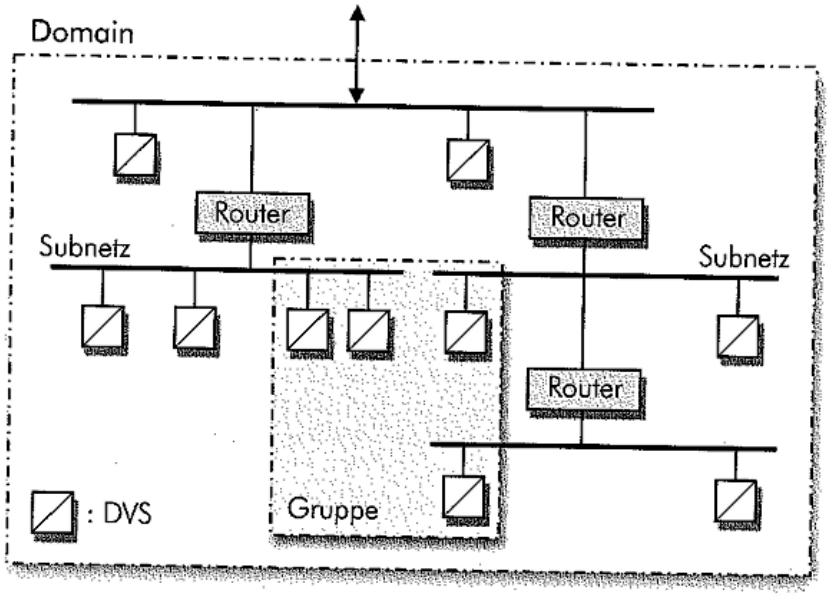


Abbildung 4.38: Baumstruktur eines LON-Bussystems

zwischen Domains ist in den LON-Standards nicht vorgesehen. Sie muss gegebenenfalls mittels Gateways vom Anwender programmiert werden.

Zugriffsverfahren

LON-Netze können bis zu etwa 1000 Anschlussstellen haben, die Ausdehnung der Netze ist beispielsweise bei einem 40stöckigen HHochhaus erheblich. Da die Zugriffszeiten zu groß wären, kann der Zugriff auf das Übertragungsmedium weder nach dem Master-Slave-Prinzip noch nach dem Token-Passing-Verfahren erfolgen. LON arbeitet daher mit einem modifizierten CSMA-Zugriff

- **Predictive p-persistent CSMA** Die Modifikation trägt der Tatsache Rechnung, dass beim ursprünglichen Ethernet die Wahrscheinlichkeit von Kollisionen bei hoher Netzbelastung stark ansteigt und sich dadurch teilweise unerträgliche Verzögerungen von Botschaften ergaben. Bei Ethernet wird der wiederholte Sendebeginn der beiden kollidierten Botschaften nach Zufallsgesetzen verzögert, wodurch bei Hochlast die Häufigkeit von Kollisionen weiter ansteigt.

  LON versucht, das Auftreten von Killisionen dadurch zu vermeiden, dass der Start einer neuen Übertragung nach Beendigung der gerade laufenden von vornherein nach Zufallsgesetzen verzögert wird, wobei die momentane Netz-Belastung in die Dauer der Verzögerung eingeht. Diese Strategie heißt "Predictive p-persistent CSMA".

  Es existiert eine Basiszeit, "Slot" genannt. Die Wartezeit TW einer neuen Botschaft nach Freiwerten des Netzes errechnet sich aus

$$TW = p \cdot nSlots \tag{4.14}$$

  mit $p = [1, 16] \in N$, durch einen Zufallsgenerator ermittelt
  mit $n = [1, 64] \in N$, abhängig von der Netz-Belastung

  Problematisch ist die Netzbelastung zu ermitteln. Man greift dabei auf die Anzahl der Quittungen zurück, die für ein ausgesandtes Telegramm bei einer DVS erwartet werden: je größer diese ist, desto höher ist die Netzbelastung (daher die Bezeichnung "predictive")

- **Verwendung von Prioritäten** Die beschriebene Modifikationen des CSMA-Zugriffs gegenüber der Ethernet-Strategie bringen sicher gegenüber dieser einen besseren Botschaften-Durchsatz, verringern die Anzahl der Kollisionen und reduzieren hierdurch die mittlere Übertragungszeit für die Telegramme. Angesichts der Vielzahl von Faktoren, die zu berücksichtigen sind, muss dennoch trotz der Verbesserung im Zeitverhalten gegenüber Ethernet festgehalten werden, dass auch LON vom Prinzip her nicht realzeitfähig ist.

Schichtenmodell

Aufgrund der Baumstruktur und der Gruppenbildung, die bei LON möglich sind, legt LON nicht nur die Schichten 1, 2 und 7 des Referenzmodells fest, sondern realisiert auch in Teilen die dazwischen liegenden Schichten. Die Gesamtheit der standardisierten Sprachelemente, die zu einem großen Teil von den Neuron-Bausteinen gehandhabt werden wird als "LONTalk" bezeichnet.

4.6.6 Interbus-S

Entstehung und Bedeutung

INTERBUS-S wurde seit Mitte der 80er Jahre von der Firma Phoenix Contact entwickelt und erstmals auf der Hannover-Messe 1987 vorgestellt. Ziel der Entwicklung war zunächst, einen möglichst einfachen Anschluss von Sensoren und Aktoren an SPSen ermöglichen, die damals auch zunehmend Aufgaben der analogen Signalverarbeitung übernahmen. Es wurde aber von vornherein ein offener Sensor/Aktor-Bus angestrebt, also nicht nur ein firmeneigener Standard.

INTERBUS-S hat Ring-Topologie; jede angeschlossene Station wirkt daher wie ein Repeater. Hierdurch sind zwischen zwei Teilnehmern Leitungslängen von 400 m und insgesamt eine Leitungslänge des Bussystems von 13 km möglich. Bei einer Übertragungsrate von 500 kbit/s ist dies im Vergleich mit anderen Feldbussystemen ein beachtlicher Wert. Als Übertragungsverfahren kommt der Manchester-Code zum Einsatz.

Das Angebot an Zubehör für den INTERBUS-S ist heute ähnlich vielgestaltig wie für die anderen Feldbus-Systeme: man erhält Buskabel, ASICs für die Busanschlüsse, Einsteckkarten für alle gängigen PCs und SPSen ebenso wie die zugehörige Anwendungs-Software, Software-Entwicklungs- und Inbetriebnahmewerkzeuge.

Buszugriff mit Summenrahmen-Verfahren

Der Character eines Sensor/Aktor-Busses zeigt sich aber nach wie vor beim Zugriffsverfahren auf das Übertragungsmedium, das durch den Begriff Summenrahmen-Verfahren gekennzeichnet ist. Dessen Prinzip wird anhand des Beispiels im folgenden Bild beschrieben.

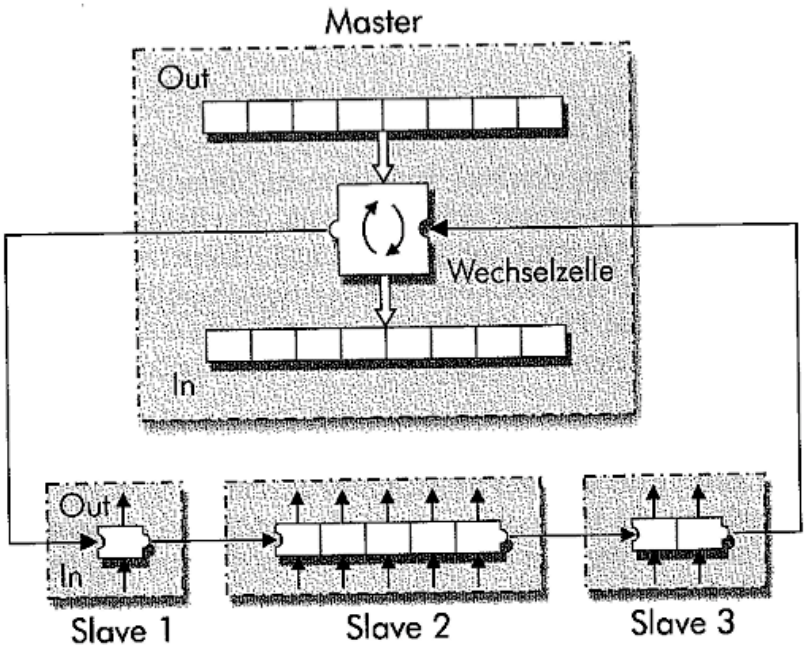


Abbildung 4.39: Zum Prinzip des Summenrahmen-Verfahrens

INTERBUS-S arbeitet in einer Ring-Topologie nach dem Master-Slave-Prinzip, allerdings in einer sehr speziellen Ausprägung. Diese Ausprägung ist wie bei einer SPS durch einen Zyklus gekennzeichnet, der allerdings vor allem in wesentlichen Teilen mittels Hardware-Komponenten realisiert wird. Bei einem Master (mehr sind im Netz nicht vorgesehen) sind bis zu 512 Slaves möglich.

- **Master** Als Master fungiert üblicherweise ein Industrie-PC mit Prozessperipherie oder eine SPS auf der B&B- oder der Feld-Ebene; hierfür hat sich in der Sprache der Feldbussysteme die Bezeichnung PNK (ProzessNahe Komponente) eingebürgert. Dieser Master sendet einen Summenrahmen aus, der alle Bytes umfasst, die von allen Slaves während eines Zyklus benötigt werden. Hierfür steht ihm eine "Wechselzelle" zur Verfügung, aus der er während eines Zyklus die von den Slaves einlaufenden Bytes für eine Weiterverarbeitung entnimmt und durch die bei ihm für den nächsten Zyklus ermittelten Bytes ersetzt, die an die Slaves gesendet werden sollen. Die zu sendenden Bytes werden aus dem Speicher der PNK entnommen, die aus der Bytekette entfernten dort zur Weiterverarbeitung abgelegt.

- **Slaves** Jeder Slave verfügt über ein Schieberegister, dessen Länge der Anzahl der Bytes aus dem Summenrahmen entspricht, die für ihn von Bedeutung sind. Im Beispiel sind dies für die Slaves 1, 2 und 3 also jeweils 1, 5 und 2 Bytes. Zu einem bestimmten Zeitpunkt des Zyklus entnehmen die Slaves die für sie bestimmten Nachrichten aus ihrem Schieberegister ("Out") und ersetzen sie durch ihre aktuellen Botschaften an den Master ("In"). Ein Reglerbaustein kann also quasi gleichzeitig den aktuellen Istwert der Regelgröße an den Master senden und im selben Zyklus den neuen Sollwert empfangen.

Um den Zeitpunkt zu erkennen, zu dem der Datenaustausch eines Bytes zu geschehen hat, werden neben den im Bild ausschließlich dargestellten Nutz-Bytes auch noch Steuerbits übertragen. Dies geschieht entweder auf zur Nutznachricht parallelen Leitungen oder dadurch, dass die Steuerbit-Folgen zusammen mit der Nutznachricht ein vereinbartes Gesamt-Telegramm bilden.

Systemsicherheit

Die Datensicherheit wird in erster Linie durch eine Prüfsummer gewährleistet, die vom Master durch einen 16-Bit-CRC am Ende des von ihm erstellten Gesamttelegrammes ermittelt wird. Da aber jeder Slave Daten aus der Byte-Kette entnehmen und andere dafür einfügen kann, muss jeder Slave eine neue Prüfsumme berechnen und an den nächsten Slave in der Kette bzw. an den Master weitergeben. Wenn ein Slave einen Fehler feststellt, teilt er dies dem Master in einem Control-Word mit, das genau so wie der Telegramm-Header an den Schieberegistern für die Nutzdaten vorbei läuft.

Durch die Ring-Struktur von INTERBUS-S und den in den Stationen realisierten Schieberegistern besteht natürlich die Gefahr, dass das gesamte Feldbussystem zum Erliegen kommt, wenn eine Komponente ausfällt. Aus diesem Grund wurde eine Watchdog.Funktion vorgesehen: Wenn über die Dauer von 20 ms keine Aktivität auf dem Bus vorhanden ist, wird dies von den Stationen als Ausfall des Systems angesehen, und sie schalten dann in einen sicheren Zustand. Wenn sich aber semantisch gerade nichts im Bus ereignet, schiebt der Master zwischen die Nutznachricht sogenannte Statustelegramme ein, die aus BBitmustern ohne Informationsgehalt bestehen.

Ein Vorteil des Summenrahmen-Verfahrens ist, dass die übermittelten Daten jederzeit konsistent sind: zu keinem Zeitpunkt außerhalb des Zeitrahmens eines Zyklus können die Werte einer Variable von einander verschieden sein.

4.6.7 ASI

ASI steht für Aktor-Sensor-Interface. Dieses Kommunikationssystem wurde in den Jahren 1990/93 von mehreren deutschen Herstellern als Sendor/Aktor-Bus entwickelt. Er unterscheidet sich vom Prinzip her nicht von einem Feldbussystem, sollte aber sehr einfach zu handhaben und sehr kostengünstig sein. Er wird als Beispielsystem am Anfang dieses Kapitels beschrieben.

4.6.8 Weitere Feldbusse

WorldFIP, P-NET und Foundation Fieldbus

- **WorldFIP** (FIP ist das Acronym für Factory Information Protocol) entstand Anfang der 90er Jahre des 20. Jahrhunderts unter Federführung der französischen Firma Telemecanique durch eine Zusammenarbeit von 120 französischen und US-amerikanischen Hersteller- und Anwenderfirmen. Die Merkmale der zunächst rein französischen Version von FIP sind in [20] beschrieben.

- **P-NET** wurde zunächst in Dänemark als ein Sensor/Aktor-Bus entwickelt, wurde dann aber auch in größeren Automatisierungsprojekten eingesetzt. Wie bei PROFIBUS werden das Token-Passing- und das Master-Slave-Prinzip mit einander kombiniert. Das Weiterreichen des Tokens ist allerdings anders als bei PROFIBUS organisiert ([20]).

  Die beiden Feldbussysteme, die neben PROFIBUS in der Euronorm EN 50 170 standardisiert sind, haben in Deutschland nur eine relativ geringe Verbreitung und werden daher hier nicht weiter behandelt. Dies erscheint auch dadurch gerechtfertigt, da sich die Merkmale von PROFIBUS FMS mit denen dieser beiden Bussysteme in wesentlichen Teilen überdecken.

- **Foundation Fieldbus** Merkmale aller drei Feldbussysteme finden Eingang in die Standards des "Foundation Fieldbus", der ebenfalls in die genannte Euronorm aufgenommen werden soll. Hinter diesem Bussystem steht die Fieldbus Foundation, die 1994 durch Zusammenschluss des ehemaligen Interoperable Systems Project (ISP) der Firma Siemens, Yokogawa, Rosemount und Fisher Control mit der US-amerikanischen Sektion von WorldFIP entstanden ist. WorldFIP war international ein sehr starker Konkurrent von PROFIBUS FMS, so dass der Zusammenschluss zu einem Gremium, das eine weltweite Standardisierung eines Feldbusses durchsetzen konnte, nur logisch war.

EIB

EIB steht für European Installation Bus. Es ist eine im europäischen Rahmen standardisiertes Bussystem ausschließlich mit dem Ziel der Gebäude-Automation, also zur selbsttätigen Steuerung von Heizungen, Lüftungen, Jalousien, Rollläden, angeschlosssenen Elektrogeräten usw., aber auch zur Überwachung, ob beispielsweise alle Fenster geschlossen sind. Es erhebt den Anspruch, auch für den Installateur verständlich zu sein.

EIB könnte längfristig auf dem Teilgebiet der Automation von Gebäuden eine ernst zu nehmende Konkurrenz zu LON werden, aber eben nur dort. Bisher stehen die zu hohen Kosten einer weiteren Verbreitung im privaten Bereich im Wege. Dies kann sich natürlich wegen steigender Ansprüche der Öffentlichkeit an den persönlichen Energieverbrauch bei Neubauten durchaus ändern.

DIN-Messbus

Der DIN-Messbus entstand aus der Zusammenarbeit mehrerer Hersteller von Automatisierungsgeräten, der Phsikalisch-Technischen Bundesanstalt (PTB) und verschiedenen Hochschulen. Ziel der Entwicklung war, ein offenes, genormtes Feldbussystem zu schaffen. Durch Verwendung von Vierdraht-Leitungen und durch klare Vorschriften für eine galvanische Trennung sämtlicher angeschlossener Geräte mittels Transformatoren, Optokopplern und DC/DC-Wandlern wird eine hohe Störsicherheit der Datenübertragung erreicht.

4.7 Other (more or less popular) communication interfaces

4.7.1 Ethernet

Originally released in 1980, Ethernet has become the standard network of computer systems worldwide. There is now a general acceptance of Ethernet as a computer interface in data aquisition/measurement. Theories abound explaining Ethernet's slow migration into data acquisition, perhaps the most common is that previously many engineers felt Ethernet systems were too difficult to configure and only trained IT personnel should dare. Of course, as the technology advanced, things got simpler, and today most teenagers are perfectly capable of installing a LAN in their houses and even most of us old-timers will have an easier time setting up a network than programming the VCR! Standard Ethernet's 100 Mbps data transfer rate is fast enough for all but the fastest data acquisition applications and its 100-meter range is also sufficient for the vast majority of systems. Ethernet systems can also be quite portable, since the only tie required to the host computer is a CAT5 cable. Ethernet-based systems are easily expanded as ports may be added with extremely low cost, off the shelf routers. However, users should be careful to keep track of total system bandwidth requirements as all of the devices on a single Ethernet port share the bandwidth. Ethernet ports are included on virtually all computers sold these days and most evidence points to this continuing for the foreseeable future. The IEEE has worked very hard to maintain backward compatibility among Ethern revisions and so even as the Ethernet specification progresses, Ethernet equipment purchased today should be useful for many years to come. Ethernet communication is generally considered very secure and is therefore used by some of the largest manufacturing and office facilities. Inter operability of Ethernet based data acquisition devices from multiple vendors has not always been stellar. However, most Ethernet based data aquisition (as opposed to Instrument) systems are single vendor and this has not been a major issue in the data acquisition space. The LXI Consortium has developed a specification that ensures simple and seamless multi-vendor interoperability.

GigE

As the name implies, Gigabit Ethernet is a version of Ethernet that supports 1 Gigabit per second data transfer rates. Other Ethernet specifications, such as deployment range and data types, remain unchanged. One thing to note is that to take advantage of the Gigabit bandwidth, your system needs to be developed accordingly. This means using either Cat5e or Cat6 cables, as well as a adding a Gigabit port to your computer and Gigabit routers/switches. Gigabit is still a new technology, so most Ethernet-based DAQ products do not yet support the faster bandwidth (and in many, if not most, applications, the extra bandwidth is not required). However, many new computers' standard Ethernet interface/ports are now 1000Base-T capable and low cost, off-the-shelf Gigabit routers/switches are also available. Ultimately, most networks of the future will probably be developed as Gigabit, but in most cases there may be little reason to

retrofit existing networks or installations.

4.7.2 Serial Interfaces

RS-232, RS422/423/485 (V.24)

People first began predicting the demise of RS-232 in the 1980s. Of course, RS-232 is still around and kicking. The RS-series ports remain extremely common in the data acquisition and control arena. RS-232 is older, and slower than its 422/423/485 family mates, but usage of both is still very common. As a fairly simple interface, there is not too much to consider when specifying an RS-series interface, but a few words may be in order. First, not all serial devices operate at the same speed. Be sure to specify a device that will handle the baud rate of your device. Second, for stable and consistent operation, especially at higher speeds, be sure to select a device with a substantial FIFO. Note that RS-232 ports, and in particular those on older devices, use hardware handshaking signals such as ”Ready to Send”, ”Clear to Send”. Many newer RS-232 interfaces do not support these handshaking signals, so be sure to check that your serial interface supports what you need. Another common series of questions arise when considering the differences between RS-422, 423 and 485. RS-422 uses a two-wire, fully differential signal interface. RS-423 uses the same signal levels, but uses only one of the two wires. RS-422 and RS-485 are almost identical. The difference is that an RS- 485 is networkable and can be connected to multiple serial devices. An RS-485 interface will almost always be perfectly suitable for talking to an RS-422 device.

USB

USB has totally supplanted RS-232 as the communications interface of choice for most consumer items such as printers, cameras and the like. Though the initial releases (v1.0 and 1.1) were slow for a number of applications, version 2.0 has all the bandwidth most DAQ applications will ever require. Virtually every new computer includes multiple USB ports. It has also become very popular in the DAQ marketplace and there are a large number of vendors offering USB-based DAQ. USB’s simple plug-and-play installation, combined with its 480 Mbps data transfer rate, make it an ideal interface for many data acquisition applications. Also, the popularity of USB in the consumer market has made USB components very inexpensive. New, low cost, USB DAQ devices are now available at previously unheard-of prices USB’s 5-meter range is perhaps its largest detraction as it limits the ability to implement remote and distributed I/O systems based on USB. There is also concern among some in the industrial arena that the USB connection’s lack of a locking cable mechanism might allow a USB cable to vibrate out of its connector. To operate USB, a USB stack is needed. Because of the universal design and flexibility, such a USB stack is rather complex, and a typical implementation is in the range of 10-30 KBytes of code, making it a non-trivial implementation. Additionally the variances on the host side despite the standard definition makes it challenging. For this reason, are often offered by the manufacturers drivers (represents a black box) only for Windows operating systems. This limits the possibility of ”universal” utilization and therefore is in productive distributed systems difficult to use.

Firewire, IEEE-1394a/b

Initially developed by Apple Computer (with support from others), Firewire is a high speed serial interface. The Firewire specification is maintained by the IEEE and is known as IEEE-1394. The original spec, released in 1995, supported 400 Mbps transfers and is also known as Firewire

400. In 2002, IEEE-1394b was released and supports data transfer rates up to 800 Mbps (a.k.a. Firewire 800). The ”b” version also extends the maximum distance between devices. Though the distance extends beyond the original 4.5 meters, the maximum data transfer rate is reduced. The original target markets for Firewire were video and audio products. In these areas, Firewire has been very successful and has a significant market share. Firewire also has the basic requirements to make it an excellent backbone for data acquisition systems. However, at approximately the same time Firewire was being promoted, USB was coming on line. It appears that USB has ”won” the battle for data acquisition though the reasons are not intuitively obvious. At this time, there are a wide variety of data acquisiton vendors and products actively promoting USB devices, while with a few exceptions, Firewire success has been confined to the original target market of Audio and Video.

4.7.3 Wireless

An entire article could be written on the various options of wireless technologies and how they **could** apply to data acquisition. However, the reality of the market as it stands right now is that the dedicated wireless-based data aquisition market is still very small. There are a number of vendors who now offer wireless data aquisition interfaces based upon such standards as Zigbee and 802.15.4 and variants of the 802.11 standard. There are also a growing number of customers implementing systems based on standard copper Ethernet devices. Rather than using CAT-5 connections between the host system and the data acquision device, however, they are opting to use standard off-the-shelf wireless routers from firms like Linksys and Netgear.

GPIB (IEEE-488)

Originally developed by HP and designated HPIB, the GPIB bus remains the dominant interconnection standard between computers and instruments though Ethernet and USB are beginning to make inroads. However, as prevalent as GPIB is in testing and maintenance applications, it has never had a substantial impact on the data acquisition market. There are a number of GPIB data acquisition products available, but their market penetration is very small relative to PCI, PXI, Ethernet, and USB based products.

5 Echtzeitprogrammierung

5.1 Merkmale von Realzeitsystemen

Realzeitsysteme müssen auf parallel sich abspielende Vorgänge in ihrer "Umwelt" reagieren. Jeder sequentielle Vorgang im technischen Prozess wird von einem sequentiellen Programm mit spezifischen Realzeitanforderungen gesteuert. Genau genommen, wird der Vorgang von einer Task (Rechenprozess) gesteuert; nämlich die Task, die durch die Ausführung des Programmes entsteht.

Ein Realzeitsystem kann also gegenüber einem System ohne Realzeitanforderungen hinsichtlich zweier Kriterien abgegrenzt werden; nämlich hinsichtlich *Zeit* und *Parallelität*. In einem System ohne Realzeitanforderung wird zu jedem Zeitpunkt ein Programm ohne Realzeitanforderungen ausgeführt, während in einem Realzeitsystem zu einem Zeitpunkt mehrere Tasks unter Einhaltung ihres jeweiligen zeitlichen Anforderungen auszuführen wären. Die Programmierung von Realzeitsystemen heisst *Realzeitprogrammierung*

Hinsichtlich Zeit

Betrachtet man ein Programm ohne Realzeitanforderungen, so beeinflusst der Zeitpunkt und die Dauer der Programmausführung das Ergebnis nicht, z.B. die Berechnung von Statistiken aus Erfassungsdaten. Hingegen beeinflussen bei einem Programm mit Realzeitanforderungen der Zeitpunkt und die Dauer der Programmausführung das Ergebnis. Die Nichteinhaltung der zeitlichen Anforderungen würde zu falschen Ergebnissen führen. Beispiel für solche Systeme sind: Automatisierung der Heizungsanlage eines Wohnhauses, Motorsteuerung in einem Kraftfahrzeug, Regelung der Bandgeschwindigkeit in einer Kamera.

Die zeitlichen Anforderungen lassen sich in zwei Klassen einteilen: Rechtzeitigkeit und Gleichzeitigkeit.

- **Rechtzeitigkeit.** Beim Eintreffen eines bestimmten Ereignisses müssen innerhalb einer bestimmten Zeitspanne die Eingabedaten abgerufen, die Ergebnisse daraus verfügbar gemach und durch die Ausgabe von Signalen (Daten) darauf reagiert werden. Erreicht z.B. ein Paket in einer Paketsortieranlage eine bestimmte Position vor einer Weiche, so muss abhängig vom Paketziel die Weiche in die entsprechende Stellung gebracht werden; und zwar bevor das Paket die Weiche erreicht hat.

- **Gleichzeitigkeit.** Realzeitsysteme müssen gleichzeitig mehrere parallel ablaufende Vorgänge im technischen Prozess automatisieren. Die Automatisierungsaufgaben müssen in dem Rechensystem so gelöst und die Ausführung der Automatisierungsprogramme so organisiert werden, dass sie einem aussenstehenden Beobachter als gleichzeitig ablaufend erscheinen. In einem Einprozessorsystem kann zu jedem Zeitpunkt nur ein Rechenprozess ausgeführt werden, daher liegt in diesem Fall nur eine scheinbare Parallelität der Rechenprozesse vor. In einem Mehrprozessorsystem mit n Prozessoren können gleichzeitig n Rechenprozesse parallel ausgeführt werden. In diesem Fall liegt eine eche Parallelität

vor. In einer Paketsortieranlage stellen zum Beispiel die Weichensteuerung für ein Paket und die Geschwindigkeit des das Pakt befördernden Fahrzeugs zwei Aufgaben dar, die gleichzeitig vom Automatisierungsrechner wahrgenommen werden müssen.

Bild Weichensteuerung

Hinsichtlich der logischen Folge (Kausalität)

- **Kausalität in einem Singletasking-System.** n einem Singeltasking-System kann zu jedem Zeitpunkt nur eine Task aktiv sein. Mit der Beendigung einer Task wird die nachfolgende Task aktiviert. De Operationen der aktien Task werden sequentiell ausgeführt und stehen in keinerlei Relation zu den Operationen der anderen Tasks. Die Kausalität beschränkt sich also auf die Operationen einer Task.

  Der Programmablaufplan einer Task stellt ihren Kontrollfluss dar, d.h. die Reihenfolge der Ausführung der Operationen der Task in Abhängigkeit von den erzeugten/erfassten Daten Die Ausführung der Operationen kann durch Wandern einer Ablaufmarke n dem zugehörigen Programmablaufplan dargestellt werden. Die Ablaufmarke beschreibt den aktuellen Stand der Programmausführung, d.h. welche Anweisung zuletzt ausgeführt wurde und welche Anweisung als nächste auszuführen ist. Betrachtet man die Abarbeitung der Programms durch die CPU, so entspricht die Ablaufmarke dem Inhalt es Programm-Counter-Registers.

  Bild Wandern der Ablaufmarke

  Das Diagramm in xxx ähnelt einem sogenannten Petri-Netz, das zur Darstellung von Kausalitäten in einem bzw. zwischen mehreren Prozessen dient.

Kausalität in einem Realzeitsystem

Realzeitsysteme sind in der Regel Multi-Tasking-Systeme. Jede Task steuert den Ablauf eines Vorganges im technischen Prozess. Die Vorgänge wirken aufeinander und erfüllen gemeinsam einen Zweck, nämlich die Automatisierung der Gesamtanlage. Die Parallelität der Vorgänge setzt zwar voraus, dass sie voneinander unabhängig sind. Da sie aber in ihrer Gesamtheit einen gemeinsamen Zwcck erfüllen, besteht eine gegenseitige – wenn auch geringfügige – logische Abhängigkeit zwischen ihnen. Die Abhängigkeit zwischen den Vorgängen überträgt sich auch auf die sie steuernden Tasks. Die Kausalität drückt sich prakisch so aus, dass die parallel ablauffähigen Operationen mehrerer Tasks miteinander synchronisiert werden müssen. Die Kausalität in einem Realzeitsystem beschränkt sich also nicht mehr auf die Operationen einer Task, sie erstreckt sich vielmehr auf die Operationen mehrerer Tasks.

Die Abhängigkeit zwischen zwei parallel zueinander ablaufenden Tasks werden in zwei Klassen eingeteilt: *Kooperation* und *Konkurrenz*.

Eine Kooperation liegt dann vor, wenn die Tasks gemeinsam eine Aufgabe erledigen, z.B. wenn die eine Task ein Werkstück teilweise bearbeitett und dann an die andere Task zur abschliessenden Bearbeitung weiterreicht. Die beiden Tasks können parallel zueinander zwei verschiedene Werkstücke bearbeiten, für die Übergabe eines Werkstücks müssen sie sich aber miteinander synchronisieren.

Eine Konkurrenz liegt dann vor, wenn die beiden Tasks gemeinsame Betriebsmittel be-

nutzen. Ein Betriebsmittel kann entweder im Besitz der einen oder der anderen Task sein. Ansonsten können die Tasks parallel zueinander ablaufen. Eine Förderstrecke in einer Paketverteilanlage stellt z.B. solch ein Betriebsmittel dar. Sie wird jeweils an eine Task vergeben, damit Pakete nicht kollidieren.

Die Synchronisierung zwischen zwei parallelen Tasks beschränkt sich auf wenige Stellen. Daher spricht man auch on lose gekoppelten Prozessen (*loosly coupled processes*). Den Gegensatz dazu hierzu bilden die eng gekoppelten Prozesse (*tighly coupled processes*). Sie kommen in Rechensystemen vor, die aus einem mehrdimensionalen Feld von parallel geschalteten Prozessoren bestehen, die über Datenkanäle miteinander verbunden sind., z.B. Digitale Signalprozessoren oder auch Opteronprozessoren mit Hyperchannel.

Synchrone und Asynchrone Programmierung

Zur Erfüllung der Anforderungen Rechtzeitigket und Gleichzeitigkeit der Programmabläufe gibt es zwei Verfahren:

- **Synchrone Programmierung**. Planung des zeitlichen Ablaufs der Teilprogramme vor ihrem Lauf. Die Planung erfolgt in der Software-Entwurfsphase. Die Teilprgramme werden in einem vorher festgelegten Zeitraster zyklisch ausgeführt.

- **Asynchrone Programmierung**. Hier gibt es keine Planung in der Entwurfsphase. Die Ausführungen der Teilprogramme stehen in keinerlei Bezug zueinander. Die Organisation des zeitlichen Ablaufs der Teilprogramme erfolgt während ihrer Ausführung (at run time), falls es zu Überlappungen kommen sollte.

5.2 Synchrone Programmierung

5.2.1 Vorgehensweise

Die prinzipielle Vorgehensweise bei der synchronen Programmierung ist wie folgt:

- Jedes Teilprogramm übernimmt die Automatisierung eines Vorganges im technischen Prozess.

- Alle Teilprogramme werden mit einem Zeitraster synchronisiert, d.h. zyklisch zur Ausführung gebracht.

- Die Reihenfolge des Ablaufs der verschiedenen Teilprogramme wird fest durch ein Steuerprogramm vorgegeben. Es entstehen also keine keine Konfliktfälle zwischen den Teilprogrammen.

- Voraussetzung für die Einhaltng der Synchronität ist, dass alle für einen Zyklus vorgesehenen Teilprogramme in dem betreffenden Zyklus beendet werden. Es darf also nicht zu Überholvorgängen kommen, die eine Abweichung von der Vorausplanung darstellen.

- Das Steuerungsprogramm wird als eine ”Interrupt Service Routine” (ISR) ausgelegt, die auf das Eintreffen des zyklischen Zeit-Interrupts zur Auführung gelangt.

Beispiele

5.2.2 Vorteile

- Einfaches Seuerungsprogramm zur Steuerung des Ablaufs der Teilprogramme.
- Garantierte Reaktions- und Antwortzeiten für alle Teilprogramme und Vorgänge im technischen Prozess.
- Ausnutzungsgrad des Rechensystems kann sehr hoch sein, da die Maximalbelastung im voraus bestimmbar ist.
- Das die Teilprogramme synchron zueinander ablaufen, bedarf die Berücksichtigung der logischen Zusammenhänge aus dem technischen Prozess keinen besonderen Aufwand. Synchronisierungen können mit einfachen Variablen und Abfragen realisiert werden.
- Wegen der Einfachheit des Steuerprogramms läßt sich die Korrektheit eines Programmsystems überhaupt nachweisen. Daher kommt bei Systemen mit extrem hoher Zuverläasigkeit (geringe Ausfallrate) oder bei sicheren Systemen (gefahrenlose Systeme) nur diese Methode in Frage.

5.2.3 Nachteile

- Die maximale Ausführungsdauer der Teilprogramme muß im voraus bestimmt werden. Dies ist aufwändig und komplizert, da die tatsächliche Ausführungszeit von Teilprogrammen Schwankungen unterliegt, und zwar wegen des dynamischen Verhaltens der Programme infolge von Abfragen und Schleifen oder wegen Schwankungen der Ein-/Ausgabezeiten.
- Einbeziehung von nicht vorhersehbaren Ereignissen (Interrupts) ist schwierig, wenn nicht sogar unmöglich.
- Die Planung des zeitlichen Ablaufs der Teilprogramme erfolgt aus der Summe der maximalen zeitlichen Anforderungen der einzelnen Teilprogramme.
- Das Steuerungsprogramm wird so ausgelegt, dass es imstande ist, nicht abhängig vom jeweiligen Prozesszustand, sondern ständig die absolut strengste zeitliche Forderung aller Teilnehmer einzuhalten. Dies ist jedoch in den meisten Fällen gar nicht nötig, so dass dann die Rechenbelastung als zu hoch eingestuft wird.
- Die zyklische Erfassung aller Prozesssignale (Polling) kann bei Ablaufsteuerungen zu sehr hohen Rechnerbelastungen führen.

5.3 Asynchrone Programmierung

Es wird versucht, die Forderungen nach Rechtzeitigkeit und Gleichzeitigkeit der Rechenprozesse zu erfüllen, ohne Voraussetzungen über den jeweilgen Zeitpunkt der Ausführung zu machen. Die dabei angewandten Grundprinzipien sind:

- Es wird zugelassen, dass Rechenprozesse "asynchron" zueinander ablaufen, d.h., zu beliebigen Zeiten in bezug auf die Echtzeit-Uhrimpulse.

- Die zeitliche Reihenfolge, in der die Automatisierungprogramme ablaufen, wird nicht fest vorgegeben.

Der asynchronen Programmierung können zwei verschiedene Modelle zugrunde liegen: quasiparalleles Modell und paralleles Model.

5.3.1 Das Modell der "Quasiparallelität"

Das Modell der Quasiparallelität liegt z.B. dem Schachspiel zugrunde und wird durch folgende Regeln characterisiert:

- Es sind zwei Spieler (zwei Prozesse) und ein Schiedsrichter (der dritte Prozess) daran beteiligt.
- Zu jedem Zeitpunkt ist nur ein Spieler am Zug.
- Zunächst übergibt der Schiedsrichter die Kontrolle an einen Spieler durch die Betätigung der Spieluhr.
- Sobald der eine Spieler geogen hat, gibt er durch die Betätigung der Spieluhr die Kontrolle an seinen Mitspieler weiter. In anderen Worten, wer im Besitz der Spieluhr (der Prozessor) ist, kann Veränderungenam Schachbrett vornehmen.
- Hat ein Spieler gewonnen, hält er die Spieluhr an und gibt die Kontrolle an den Schiedsrichter zurück. Damit werden beide Spieler-Prozesse beendet.

Merkmale des Modells der "Quasiparallelität"

- Es gibt mehrere Prozesse
- Diesem Modell liegt ein Einprozessor-System zugrunde.
- Zu jedem Zeitpunkt kann nur ein Prozess ablaufen. Die Kontrolle über die CPU hat dann dieser Prozesss.
- Der Prozess, der die Kontrolle hat, wird nicht unterbrochen und verdrängt (non preemption). Er selbst gibt die Kontrolle explizit an einen anderen Prozess weiter.
- Die Prozesse übernehmen also selbst die Kontrollzuteilungsfunktion (Scheduling, Dispatching), d.h. die Vergabe der CPU an einen Prozess.

bf Implementierung des Modells "Quasiparallelität"

Auf einem Einprozessorsystem: In diesem Fall stimmt das Modell mit der Realität überein. Die verschiedenen Prozesse der Applikationn werden einzeln in den Besitz des realen Prozessors gelangen und ihn nach Ausführung gewisser Operationen an irgendeinen anderen Prozess weiterreichen.

Auf einem Mehrprozessorsystem: Man kann sich gut vorstellen, dass eine Erhöhung der Prozessor-Anzahl bei einem Schachspiel keinen Vorteil mit sich bringt. In diesem Fall kann nur ein Prozessor des Systems tatsächlich genutzt werden. Denn es ist nicht zulässig, zwei Prozesse der Applikation in echter Parallelität zueinander ablaufen zu lassen, insbesondere wenn sie gemeinsame Daten miteinander teilen. Betrachtet man z.B. zwei Tasks , die nach dem Mo-

dell der "Quasiparallelität" konzipiert sind und über gemeiname Daten verfügen, so werden die Tasks nicht gleichzeitig, sondern hintereinander auf die Daten zugreifen. Würde man aber nun die Tasks parallel zueinander ablaufen lassen, so könnten sie doch gleichzeitig auf die gemeinamen Daten zugreifen. Dadurch können Dateninkonsistenzen entstehen.

5.3.2 Das Modell der "Parallelität"

Dem **8er-Rudern** liegt ein ganz anderes Modell zugrunde, das mit dem Modell der Parallelität bezeichnet und durch folgende Merkmale charakterisiert wird:

- Es gibt 8 Ruderer (Prozesse), die jeweils im Besitz eines eigenen Ruders (Prozessors) sind.
- Die Ruderer arbeiten parallel zueinander.
- Wären weniger Ruder (Prozessoren) als Ruderer (Prozesse) vorhanden, so würden die Ruderer mieinander um die Ruder konkurrieren und sich gegenseitig verdrängen. In diesem Fall würde eine Instanz benötigt werden, die aufgrund einer vordefinierten Strategie die Zuteilung der Ruder an die Ruderer übernimmt.

Merkmale des Modells der "Parallelität"

- Es gibt mehrere Prozesse
- Diesem Modell liegt ein Mehrprozessor-System zugrunde.
- Jedem Prozess wird ein *virtueller Prozessor* zugeteilt, so dass zu jedem Zeitpunkt alle Prozesse ablaufähig sind.
- Die Prozesse können sich gegenseitig verdrängen, wenn weniger reale Prozessoren als Prozesse vorliegen. In diesem Fall kann ein Prozess an jeder beliebigen Stelle unterbrochen und zu einem späteren Zeitpunkt fortgesetzt werden (Verdrängung, preeemption).
- Die Prozesse übernehmen selbst keine Kontrollzuteilungsfunktion (Scheduling). Dies wird von einer unabhängigen Instanz (Betriebssystem) ausgeführt.

6 Echtzeitbetriebssysteme

6.1 VxWorks

VxWorks is a real-time operating system developed as proprietary software by Wind River Systems of Alameda, California, USA. First released in 1987, VxWorks is designed for use in embedded systems.

6.2 RTEMS

The Real-Time Executive for Multiprocessor Systems or RTEMS is a open source fully featured Real Time Operating System or RTOS that supports a variety of open standard application programming interfaces (API) and interface standards such as POSIX and BSD sockets. It is used in space flight, medical, networking and many more embedded devices across a wide range of processor architectures including ARM, PowerPC, Intel, Blackfin, MIPS, Microblaze and more. You can obtain commercial support or support via the active community. Major decisions about RTEMS are made by the core developers in concert with the user community, guided by the Mission Statement. Everyone can contribute changes and help testing RTEMS. Access to the development sources is provided via a Git Repository. The provided regular, high quality releases works well on a wide range of embedded targets using cross development from a variety of hosts including GNU/Linux, Mingw, MS-Windows, FreeBSD, Cygwin, and Solaris.

7 Programmiersprachen für die Prozessautomatisierung

7.1 Definition einiger Grundbegriffe

7.1.1 Klassifizierung anhand des Programmiermodells

Wenn man Programmiersprachen anhand ihres Programmiermodells einteilt, dann werden ihre Grundkomponenten und deren Verknüpfung untersucht. Ihrem Programmiermodell entsprechend lassen sich Programmiersprachen in imperative, prozedurale, funktionale, objektorientierte und logische Programmiersprachen unterteilen.

Imperative Programmiersprachen bestehen aus einer linearen Liste von Programmbefehlen. Sie werden deshalb auch oft Befehlsfolgen-Sprachen genannt. Mit bedingten Anweisungen und Sprung-Anweisungen können diese Programmiersprachen Programmteile überspringen oder wiederholen. Typische Vertreter der imperativen Programmiersprachen sind die Sprachen FORTRAN, C oder COBOL. Bei prozeduralen Programmiersprachen werden zu lösende Probleme in Teilprobleme aufgeteilt - auch Funktionen (C/C++) bzw. Prozeduren (Modula, PASCAL) genannt. Diese Unterprogramme werden während des Ablaufs aus einem Hauptprogramm aufgerufen. Nach ihrem Ablauf wird das Hauptprogramm an der Stelle des Unterprogramm-Aufrufes fortgesetzt. Programmlogik und die zu bearbeitenden Daten werden im Programm getrennt betrachtet. Die Stärke und Schwäche zugleich von C ist seine Maschinennähe, die dem Programmierer sehr viele Freiheiten lässt und die Verwendung von maschinenorientierten Sprachen wie Assemblersprachen nahezu vollständig verdrängt hat. C wurde im Hinblick auf objektorientierte Programmierung zu C++ weiterentwickelt.

Ein Programm in einer funktionalen Sprache (z.B. LISP, Gofer, Haskell, SML) besteht aus einer Menge von Funktionsdefinitionen und einem Ausdruck, dessen Wert als Ergebnis des Programms ausgegeben wird. Diese Aufrufe können rekursiv sein; d. h., die Funktion ruft sich während ihrer Ausführung selbst auf. Die Reihenfolge der Berechnungen von Unterausdrücken wird durch die Strategie der Sprache bestimmt. Bei einer strikten Sprache werden die Argumente erst ausgewertet, bevor sie einer Funktion übergeben werden, bei einer nicht strikten Sprache werden die Argumente unausgewertet übergeben. In der funktionalen Programmierung fehlt teilweise völlig das Konzept der sequentiellen Abarbeitung. Sie ist daher deutlich anders als das prozedurale Programmieren.

Ein weiteres Programmiermodell ist die logische Programmierung. Der bekannteste Vertreter dieses Typs ist die Sprache PROLOG. Hier werden nur Fakten und Regeln angegeben. Der Weg der Problemlösung wird nicht genauer spezifiziert, sondern vom Interpreter-Programm erstellt. Mit Hilfe dieser Fakten und der angegebenen logischen Ableitungsregeln kann festgestellt werden, ob eine Behauptung zutrifft. Die zu lösende Aufgabe muss daher als eine derartige Behauptung eingegeben werden. Eine andere bekannte Sprache dieser Art ist die Query-Sprache SQL.

Moderne Sprachen sind heute meist objektorientiert (OO). Objektorientierte Programmierung

ist ein Programmiermodell. Der Ansatz ist hier ein anderer als bei prozeduralen Programmiersprachen. Im Vordergrund stehen die Daten, nicht die Funktionen des Programms. Die Daten werden in Objekten gekapselt, die auch über Funktionalitäten verfügen, um die Daten zu ändern. In diesem Zusammenhang spricht man jedoch nicht von Funktionen, sondern von Methoden. Dabei werden die Programme aus verschiedenen Objekten aufgebaut. Diese Objekte können dann über die Methoden miteinander kommunizieren. Klassische objektorientierte Vertreter sind Smalltalk, C++ und Java. Beispiele für weniger objektorientierte Sprachen sind Visual Basic und Perl. Smalltalk ist in der Umsetzung der objektorientierten Konzepte am konsequentesten.

7.1.2 Program language C

Script Programming in C, Oxford University Computing Services, 1996. [27]

code 7.1 Hello World Example

```
/* This program prints a one-line message */
#include <stdio.h>
int main() {
   printf("Hello World\n");
return 0;
}
```

compile with **make programName**. Run with **./programName**.

A table of the control characters used in printf statements is given below:

Character	Form of output	Expected argument type
c	character	int
d or i	decimal integer	int
x	hexadecimal integer	int
o	octal integer	int
u	unsigned integer	int
e	scientific notation floating point	double
f	n̈ormalf̈loating point	double
g	e or f format, whichever is shorter	double
s	string	pointer to char
p	address format (depends on system)	pointer

Arguments of type char and short int get promoted to int and arguments of type float get promoted to double when passed as parameters to any function, therefore use int conversion characters for variables of type char and short int, and double conversion characters for a float variable or expression.

More C-programming : [26]

code 7.2 Use command line arguments

```
int main(int argc, char *argv[], char *env[])
   {
int i;
   for (i = 0; i < argc; i++)
      printf("%s\n", argv[i]);
   for (i = 0; env[i] != NULL; i++)
      printf("%s\n", env[i]);
return 0;
}
```

8 Data aquisiton software/frameworks

8.1 LABView and MATLAB

LabVIEW and MATLAB were developed by two different American companies. The first is developed by National Instruments and the second by MathWorks. LabVIEW (Laboratory Virtual Instrument Engineering Workbench) is a graphical programing environment based on graphical programming language G. MATLAB (MATrixLABoratory) is the name used for the programming language and programing environment specialized for numerical calculations. Both platforms help engineers and scientists around the world in various stages of design, modeling, simulation, prototype testing, or deployment of new technologies. [22]

Introduction to Modern Data Acquisition with LabVIEW and MATLAB

By Matt Hollingsworth

Introduction to Modern Data Acquisition

Overview

Goal: To learn to use the various computer tools available to acquire and analyze experimental data.

Materials:

- Computer with LabVIEW and MATLAB
- NI USB-6008
- SummaSketchIII
- Conductive paper with electrodes
- Various cables
- Brain

This lab exercise is meant to give a sweeping overview of a couple of the tools available to a modern experimental physicist. The tools that this exercise will focus upon are LabVIEW and MATLAB.

In general, LabVIEW is a useful programming language when you need to produce some code that will acquire some data in a lab environment. You can quickly create code that will allow you to acquire some data, do some data analysis, display real-time results of that data, and export it in a format that will be capable of being read by other data analysis software (such as, in our case, MATLAB).

MATLAB, on the other hand, is a handy mathematical toolbox that comes with many features that are useful for data analysis. It is also a widely accepted industry standard, so LabVIEW comes with built-in support for directly interfacing with the script server for MATLAB. This allows you to input commands from LabVIEW directly into the MATLAB kernel and have them executed as if you were typing them in the MATLAB command window. We will be exploiting this feature in order to generate some detailed visualization of the electric field between two electrodes.

This paper will not but scratch the surface of the functionality available to you through the LabVIEW and MATLAB platforms; however, my goal is to make you conversant enough to where you know what questions to ask/documentation to search for and you can understand the answers that you are given. The more acute functionality is left for you to discover yourself as you need it.

Furthermore, if you already know something that I am discussing in the LabVIEW and/or MATLAB sections, feel free to skip it and move on to something else. There are no exercises or anything in those two sections; the most important thing is that you are able to perform the final task presented to you in the actual lab.

In case you are having trouble, each example has its corresponding .vi or .m file on my web site at http://www.evanescenthorizons.com/. To find the examples, look under Labs > Introduction to Modern Data Acquisition.

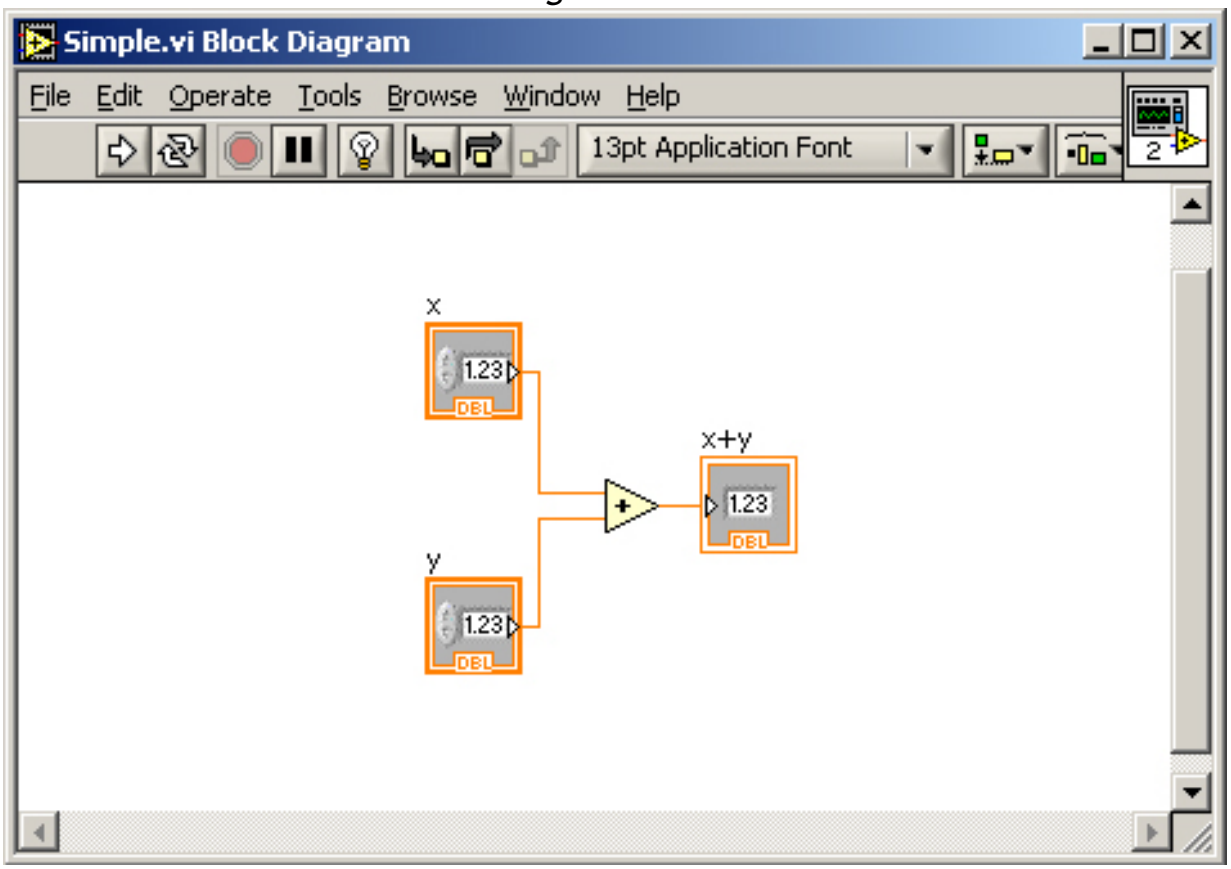
If you get stuck or have any questions, comments, complaints, or corrections, feel free to e-mail me at mhollin3@utk.edu. I'd be more than happy to help, and I welcome your feedback.

LabVIEW

Section 1.1: Introduction to LabVIEW

LabVIEW is a graphical programming language/IDE combination that is tailored for use in a lab environment. The basic analogy throughout LabVIEW is that of a virtual instrument or VI. In a LabVIEW program, just like a real instrument, you have controls (input), indicators (output), and logic to define the relationship between input and output. Here is a sample of a very basic LabVIEW program:

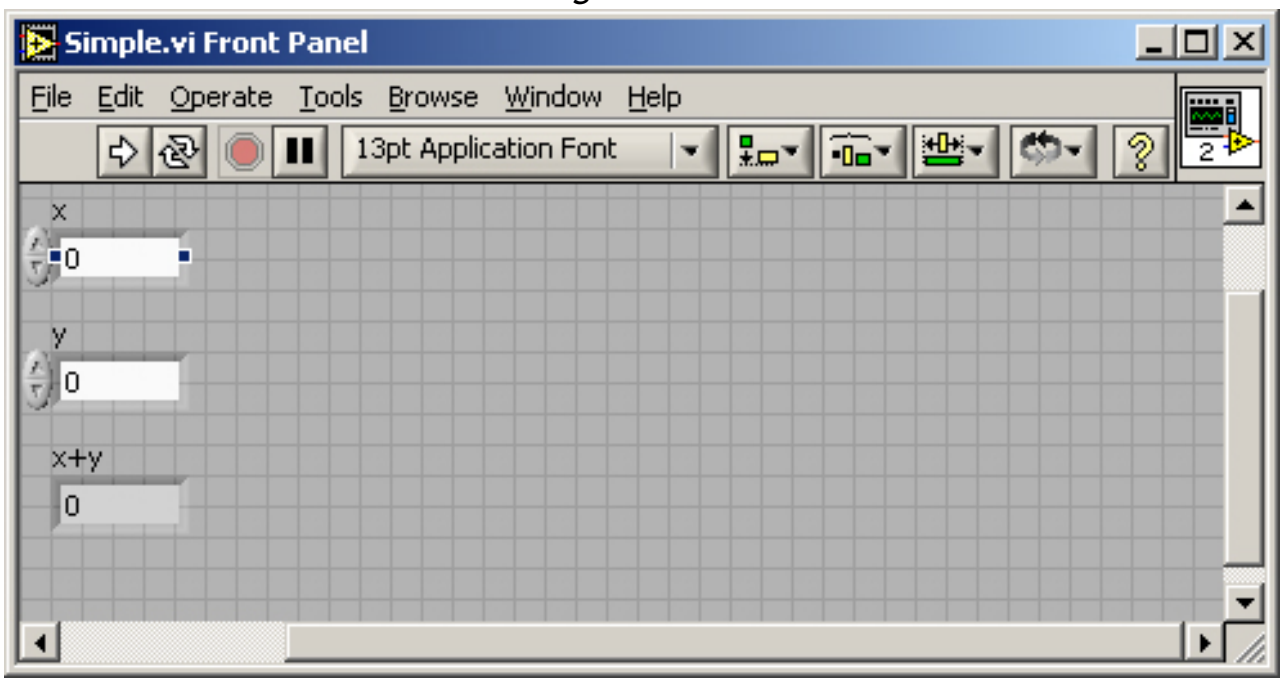
Figure 1.1.1



As you can probably already tell, this program takes two numbers, x and y, and outputs its sum. In LabVIEW, the lines that show data flow (in this case, they are thin orange lines, meaning they are single length, double accuracy numbers) are called wires. Inputs are called controls and outputs are called indicators. The plus sign inside the triangle is called a subVI, which is analogous to a subroutine or function in text-based languages. In this case, the sum subVI accepts two numbers as arguments and outputs the sum.

Logic and interface are two separate entities in a LabVIEW program. The interface is called the front panel, while the programming logic itself is called the block diagram. Here is the front panel for the program pictured above:

Figure 1.1.2



In order to run a program, you simply click the run arrow in the top left corner. To toggle looped execution, click the button with the revolving arrows (this causes the program to run continuously). The main program execution buttons are detailed below.

Figure 1.1.3 - Typical Execution Bar

 : Run Button

 : Run Continuously Button

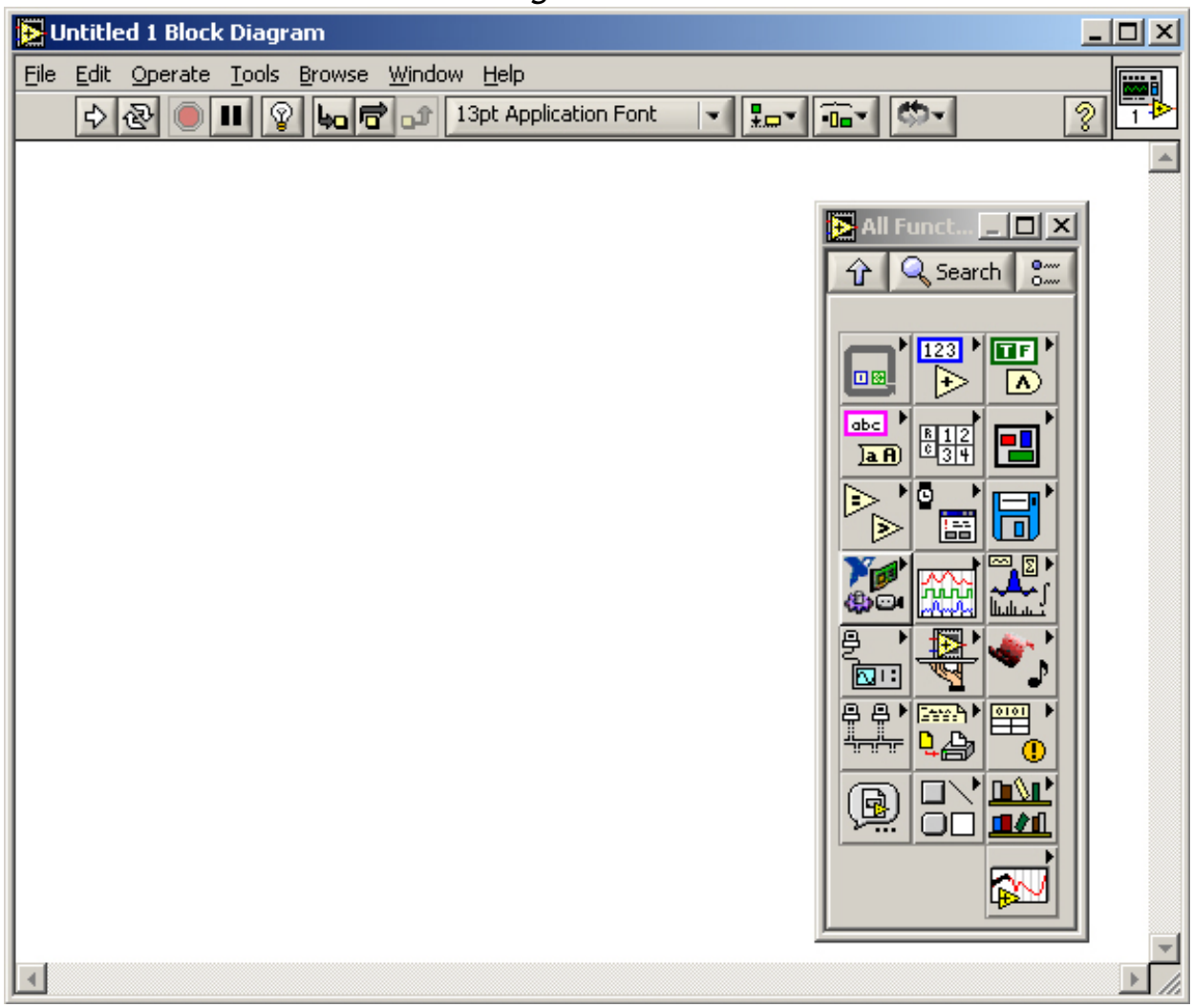
 : Abort Button

 : Pause Button

Section 1.2: A Simple LabVIEW Program

To begin writing a new LabVIEW program, simply start up LabVIEW, click new, and select blank VI when you are asked for a template. You now have a blank front panel and a blank block diagram. Navigate to the menu bar at the top of the screen, click window, and select "Show Block Diagram" (or press Ctrl + E). Right click anywhere in the white space of the block diagram to pop up the functions menu. Click "All Functions" in the bottom right of the popup window, and then click the thumbtack in the top right corner to keep the functions window open. Your screen should look something like this:

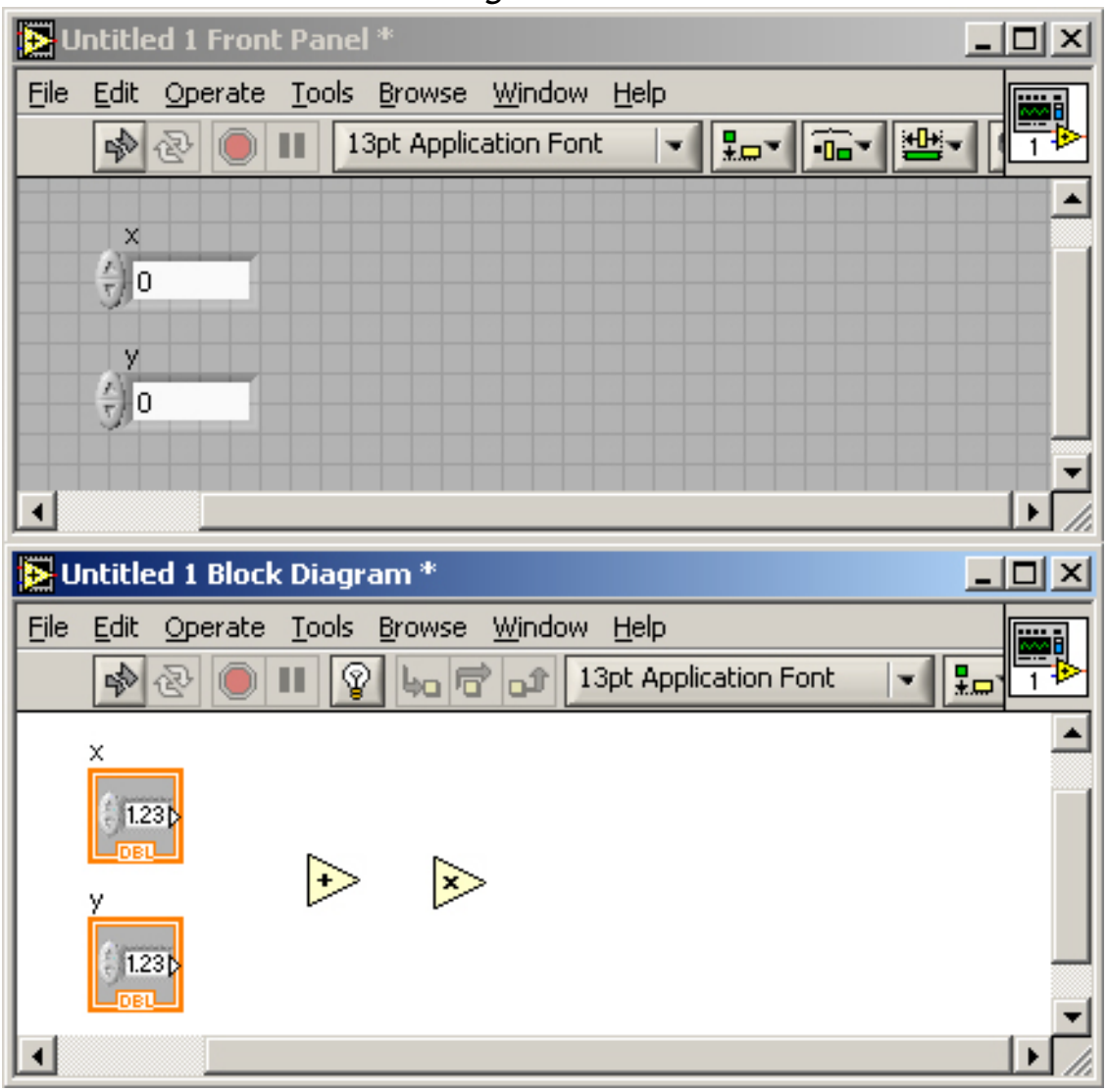
Figure 1.2.1



Click on the numeric box in the functions pallet (the on with the + and the 123 in it). Click and drag a sum subVI and a multiplication subVI onto the block diagram. Now we have some pro-

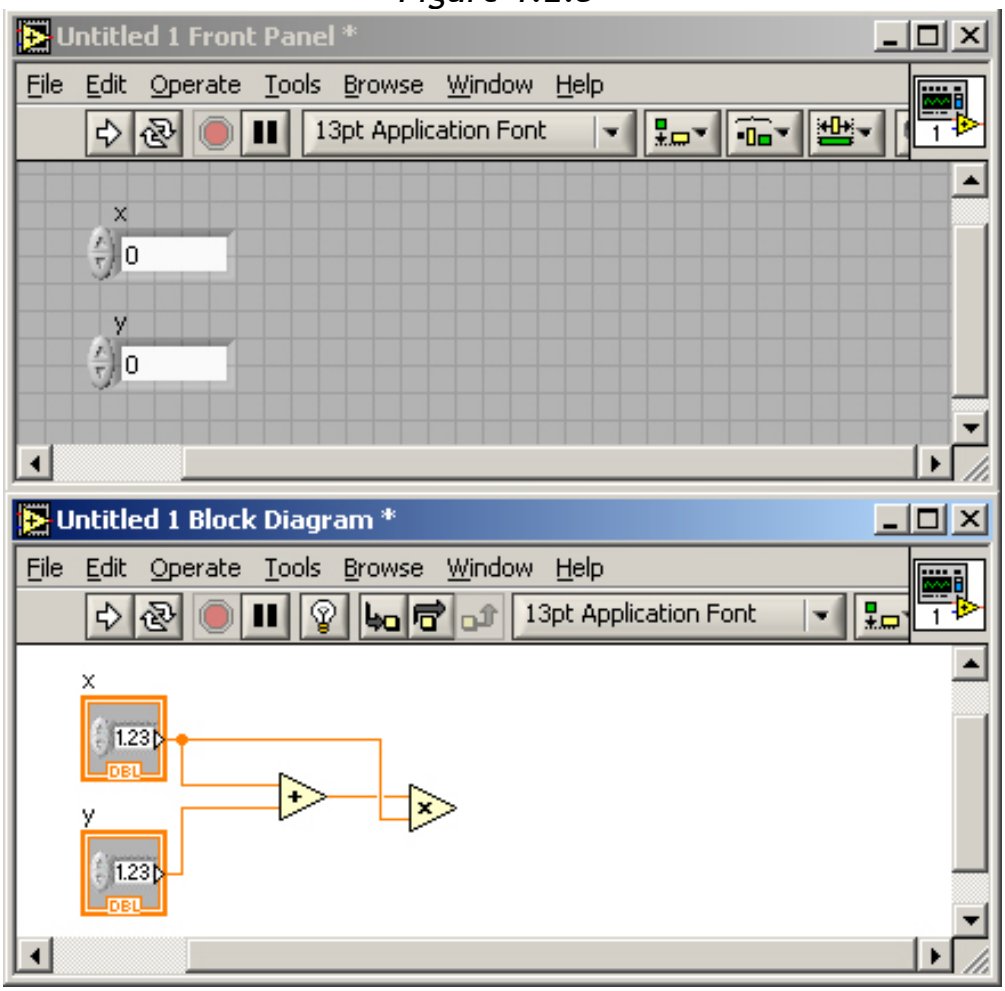
gram logic, but we need something to input data. Navigate back to the front panel by selecting Window > Show Front Panel from the menu bar at the top. Right click anywhere on the front panel, select the "Num Ctrls" box, and click and drag a "Num Ctrl" onto the front panel. Repeat so that there are two numeric controls on the front panel. Double click on the label of one of the controls ("Numeric" by default) to change the name of the control to "x". Change the other to "y". After all is said and done, your front panel/block diagram should look something like this:

Figure 1.2.2



Now, we will tell LabVIEW to output z, where z = (x+y)*x. To do this, we wire x and y to the sum subVI, wire the output of sum to one of the inputs of multiply, and then wire x to the other input of multiply, like so:

Figure 1.2.3

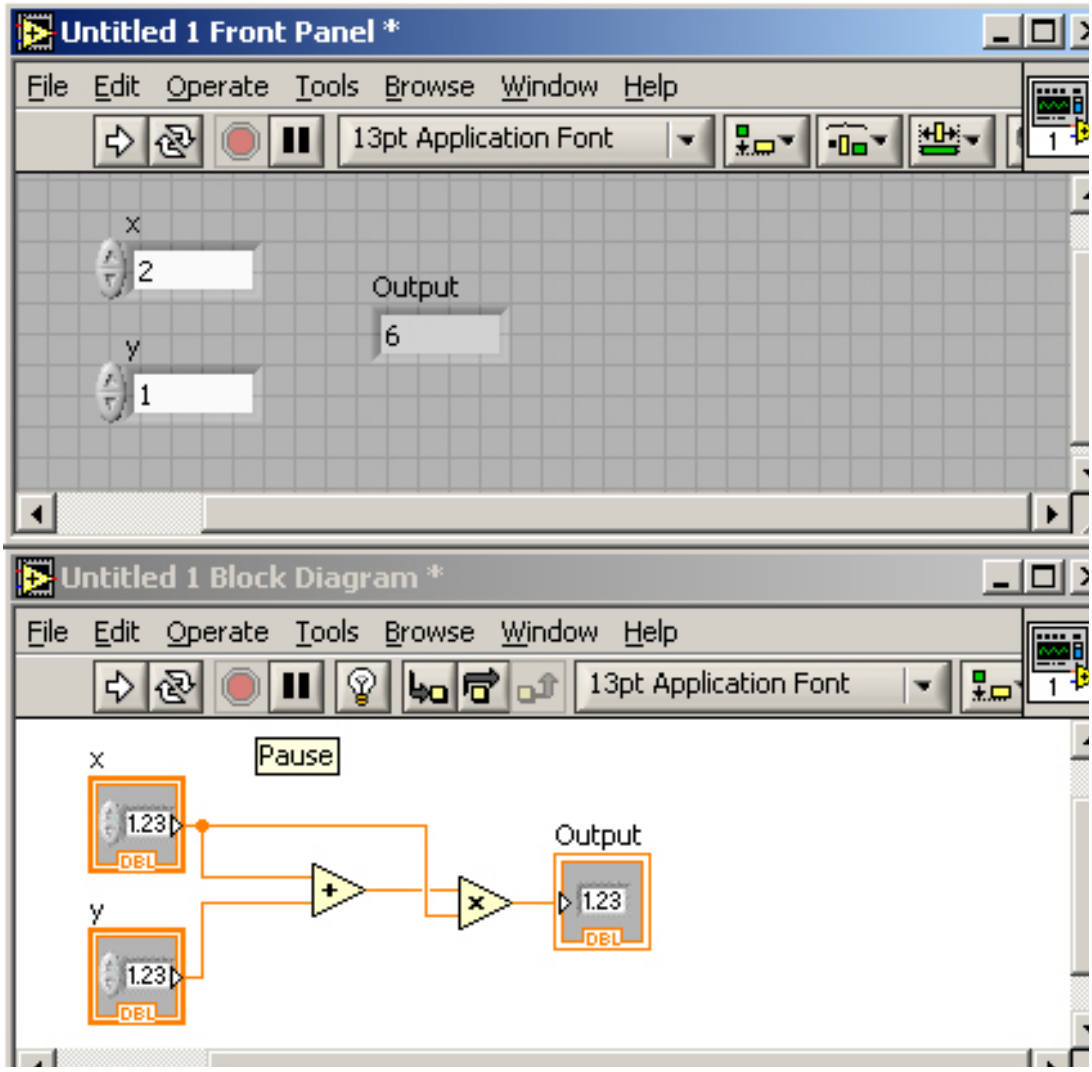


Side Note If you ever see wires appear as dotted lines with x's in various places on them, you wired something wrong. If this happens you may either right click the wire and tell it to delete the wire branch, left-click somewhere on the wire and finish wiring it properly, or hit Ctrl + B to remove all broken wires from the block diagram.

Now all that is left is to create an indicator. You can return to the front panel and create a numeric indicator the same way that you created the numeric controls, but there is a quicker way. Right click on the output of the multiplication subVI (the right most tip of the triangle). You'll see a menu pop up; select Create > Indicator. LabVIEW adds and automatically wires an indicator the output. For future reference, you may do the same thing when creating both indicators and constant terms.

Here is the completed VI:

Figure 1.2.4

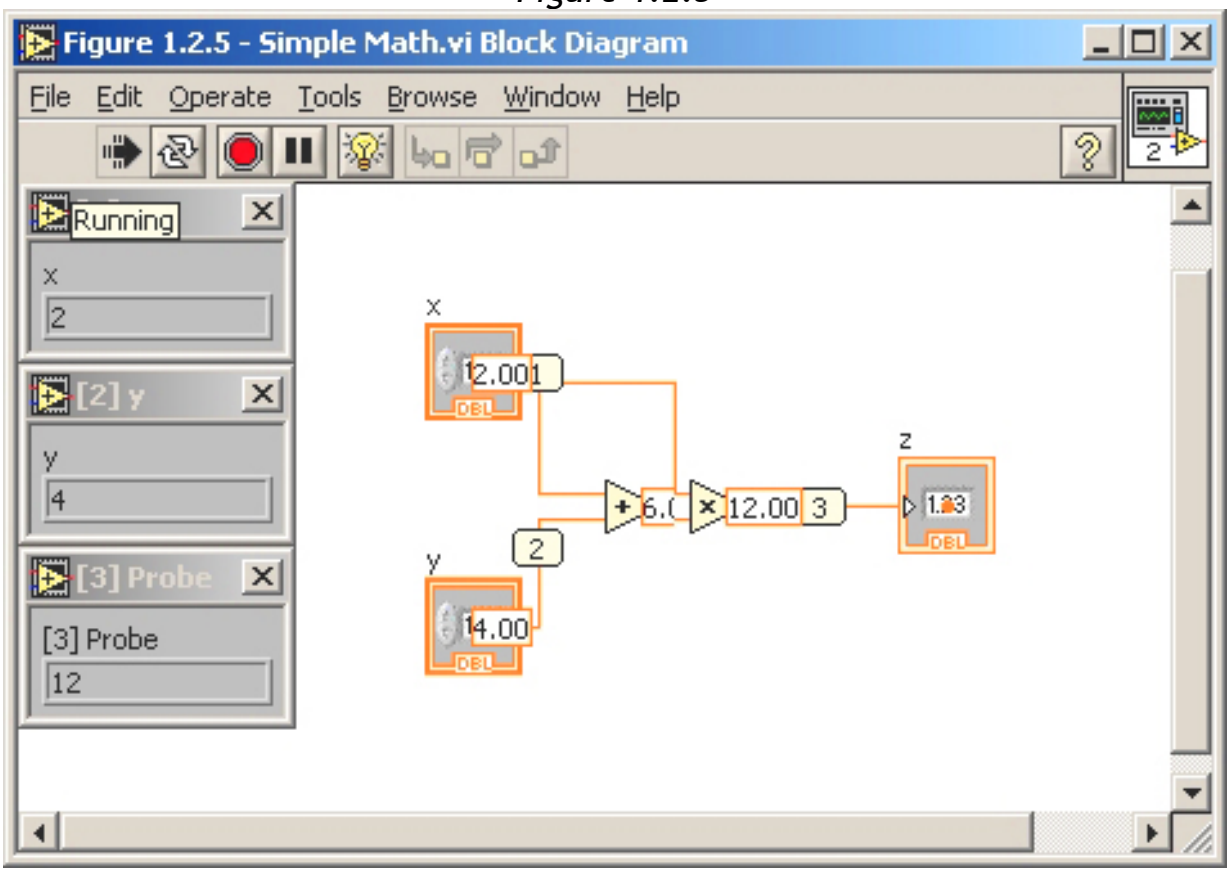


The program is now complete. Now, to test your new VI simply click the run button ( ) to run your VI. Every time that you change x/y and run the VI, you should see an output, called "output" in our program, that corresponds to output = x*(x+y)

If your program is not functioning properly, you may always click the little light bulb that is by the pause button in the menu bar of the block diagram. When this button is toggled, "Highlight Execution Mode" is turned on. This allows you to see your program execute step by step—a very useful debugging feature.

Furthermore, you may go to Window > Show Tools Palette, and place a probe on a wire by clicking the button and then clicking on a wire of which you would like to monitor the output. Here is what a probed VI with execution highlighting turned on looks like.

Figure 1.2.5



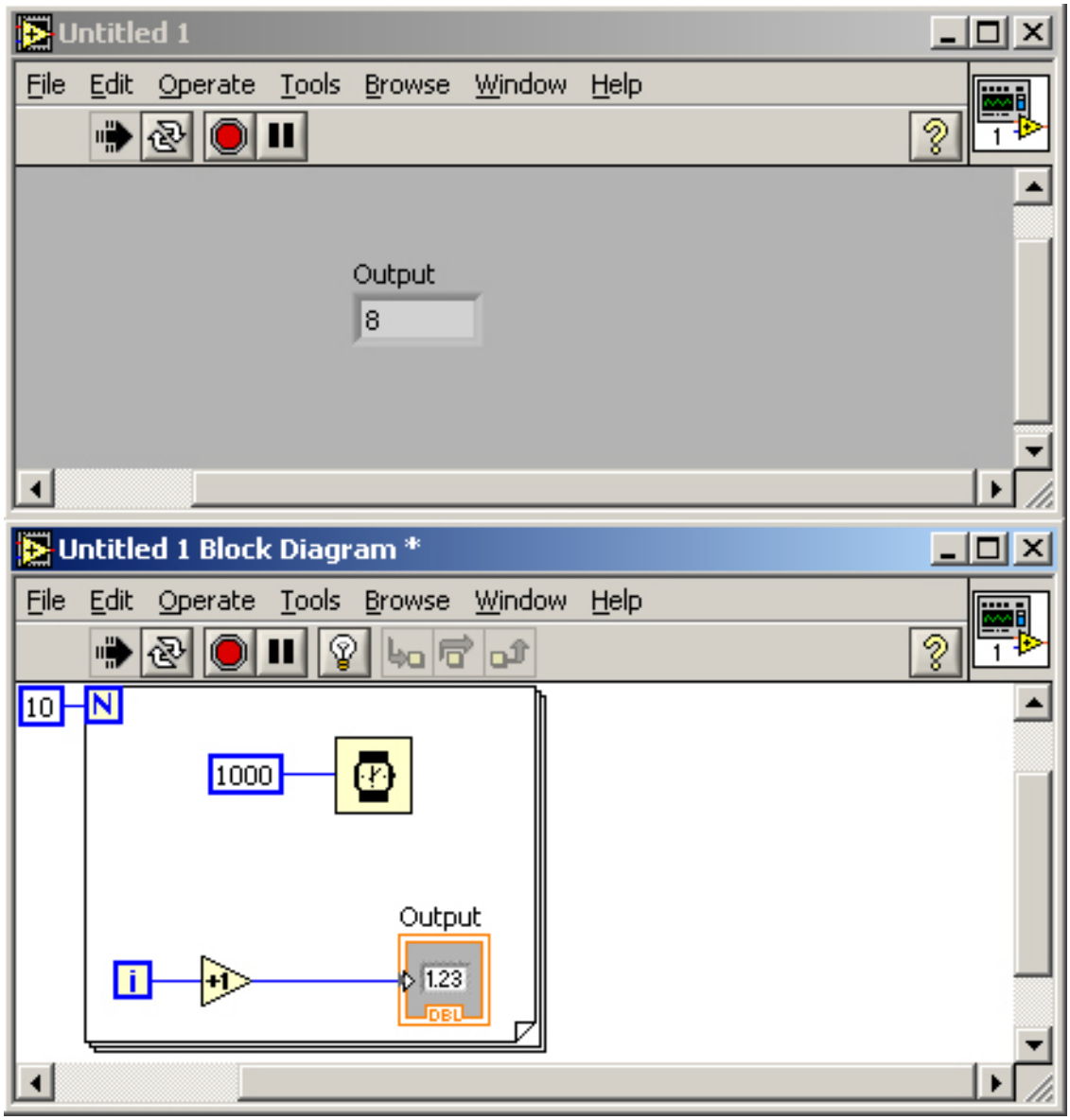
It is not necessary that every initial input be either the output of a control or the output of another subVI. If desired, you may also wire a constant to an input of a subVI. To add a numerical constant, go to the numerical palette and find the numerical constant object (the blue box with 123 in it). Put it on the block diagram. You can then right click on it and go to "Representation" to choose the data type (double precision, long integer, short integer, etc). You can then wire this constant to any numerical input of a subVI. There are other types of constants, such as array constants, enum constants, ring constants, etc., and they all work the same way.

Finally, if you ever get stuck, you should turn on the Context Help feature. To enable this feature, go to the help menu and click "Show Context Help." As long as this mode is on, whenever you hover over a subVI a window will show you help information pertaining to the usage of that particular VI. This is quite useful as a quick reference for times in which you know what a subVI does, but don't know exactly how to use it.

Section 1.3: Structures and Execution Control

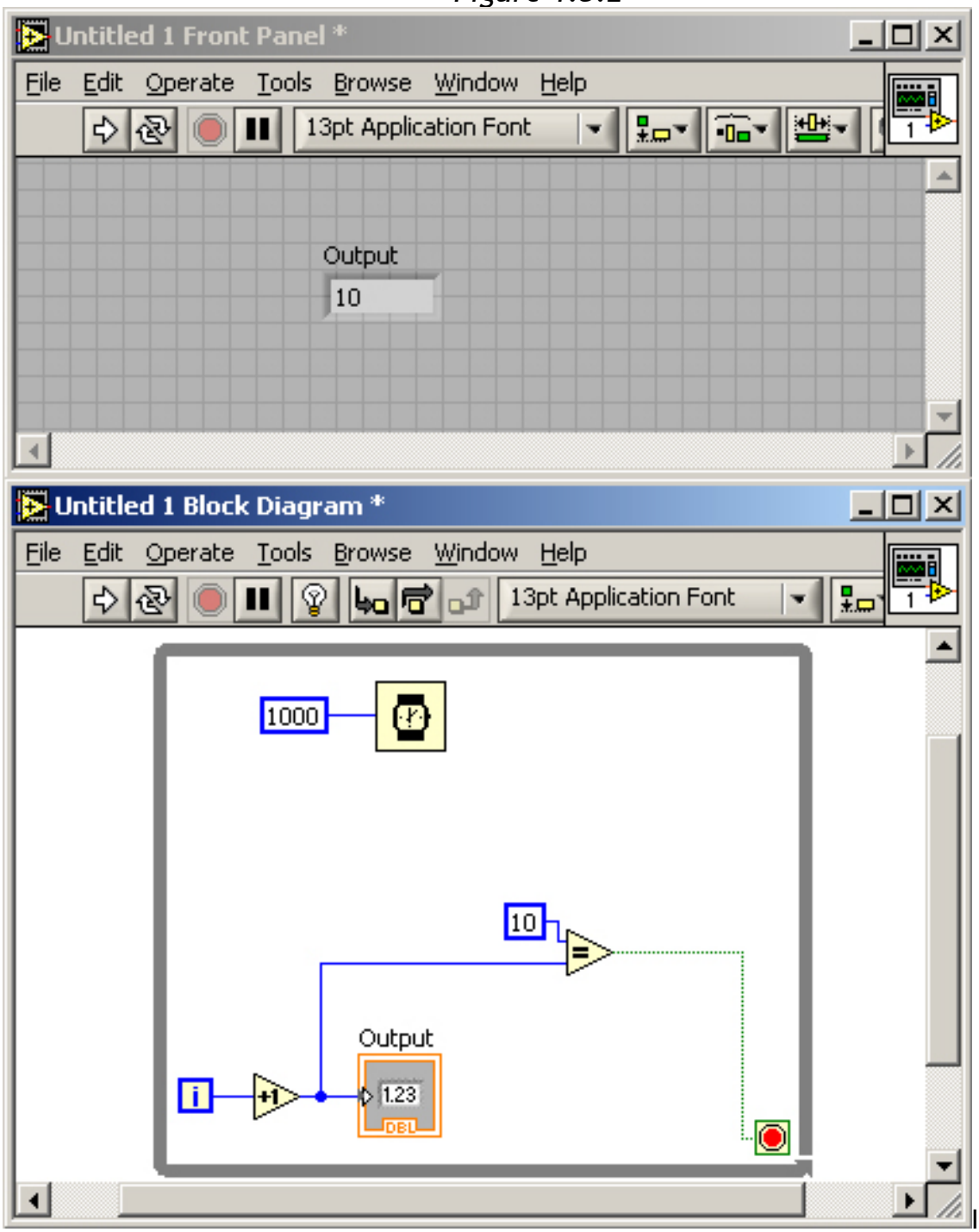
Just as with any other programming language, LabVIEW comes complete with for loops and while loops. A "for loop" loops some code a set number of times, and a "while loop" loops until a certain condition is met. In LabVIEW, loops are represented by a box that surrounds the code that is being looped. For example, the following code counts from 1 to 10 on 1 second intervals:

Figure 1.3.1



The demonstrated loop is a for loop. N is the number of iterations you would like to do, and i is the zero based index of the current iteration. An equivalent way to formulate this algorithm, using a while loop instead, can be done like so:

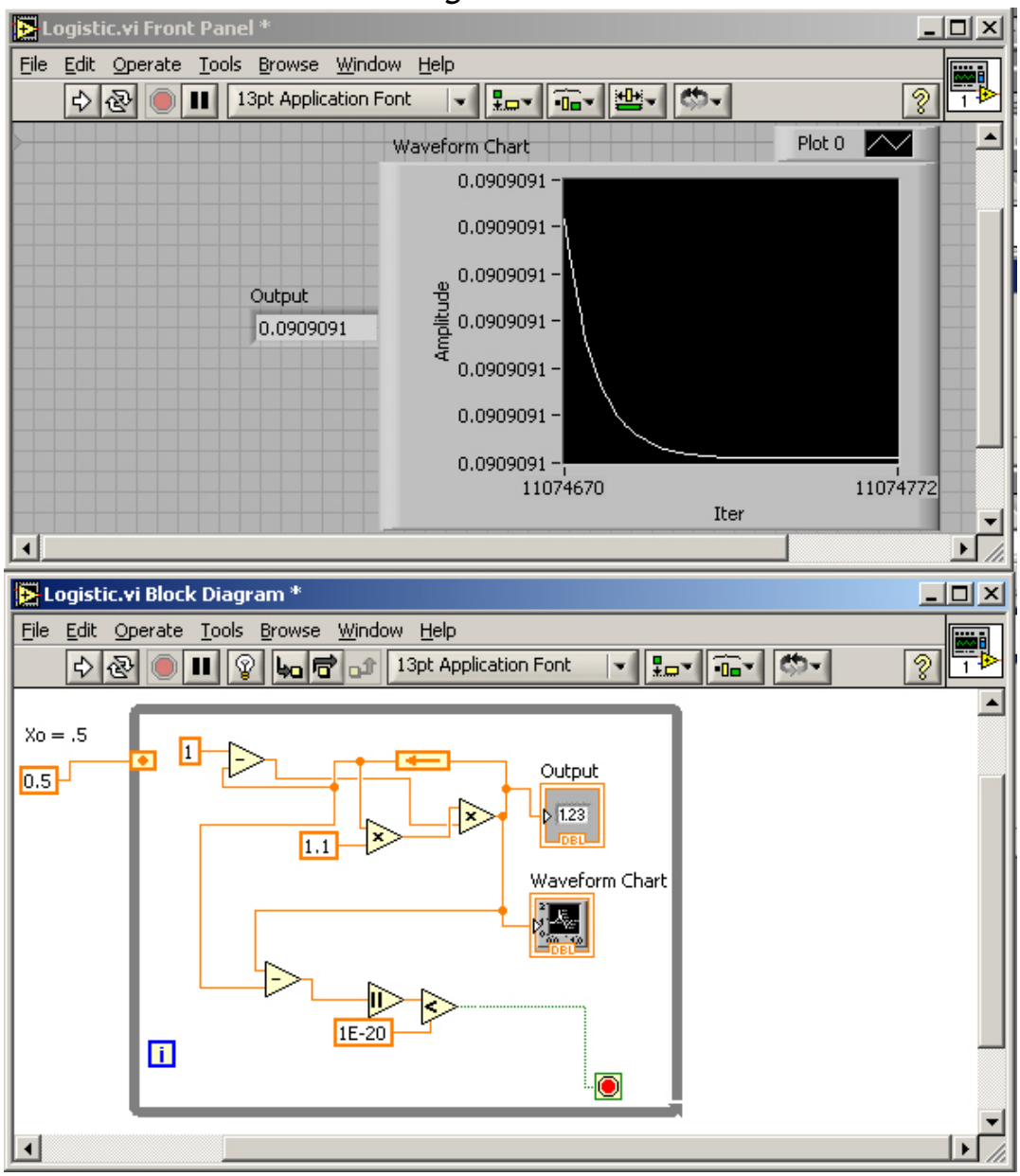
Figure 1.3.2



In a while loop, the stop sign is the termination condition. In this case, the stop condition is that the iteration index + 1 = 10, which is equivalent to a for loop with N = 10. By default, the loop terminates if the condition is true. However, you may change this to continue as long as the condition is true

by right clicking on the stop sign and selected "continue if true." This code takes the logistic map, , with alpha = 1.1 and terminates when the difference between and is less than .01. We will return to this example later for a more thorough discussion.

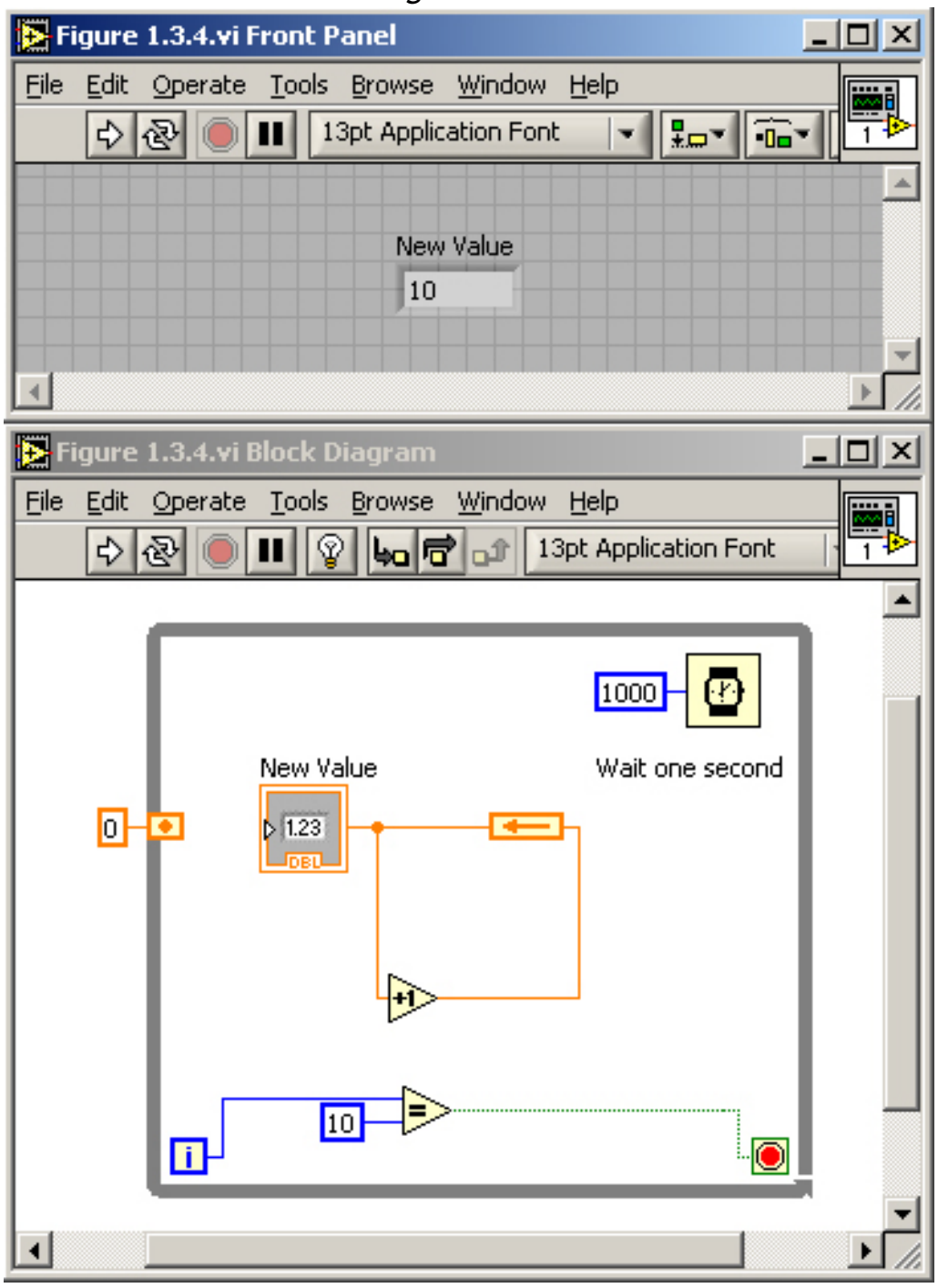
Figure 1.3.3



Now is a good time to introduce feedback nodes/shift registers. These two objects provide a way to pass data from one iteration of a loop to the next. For example, say that you want a loop to add one to a number every iteration, i.e., x0=0, x1=1….xn=n. To do this in LabVIEW, you would first wire a 0 to the initializer terminal (the terminal created on the edge of the left side of the loop that looks like a rectangle with a diamond in it). This dictates the first value that the feedback node will return. You then wire the new value to the "back" of the feedback node (mean-

ing the tail of the arrow as opposed to the head). Thus every iteration of the loop will give a new value to the feedback node, and the feedback node will report that new value when the next iteration begins. Here is an example of using a feedback node.

Figure 1.3.4

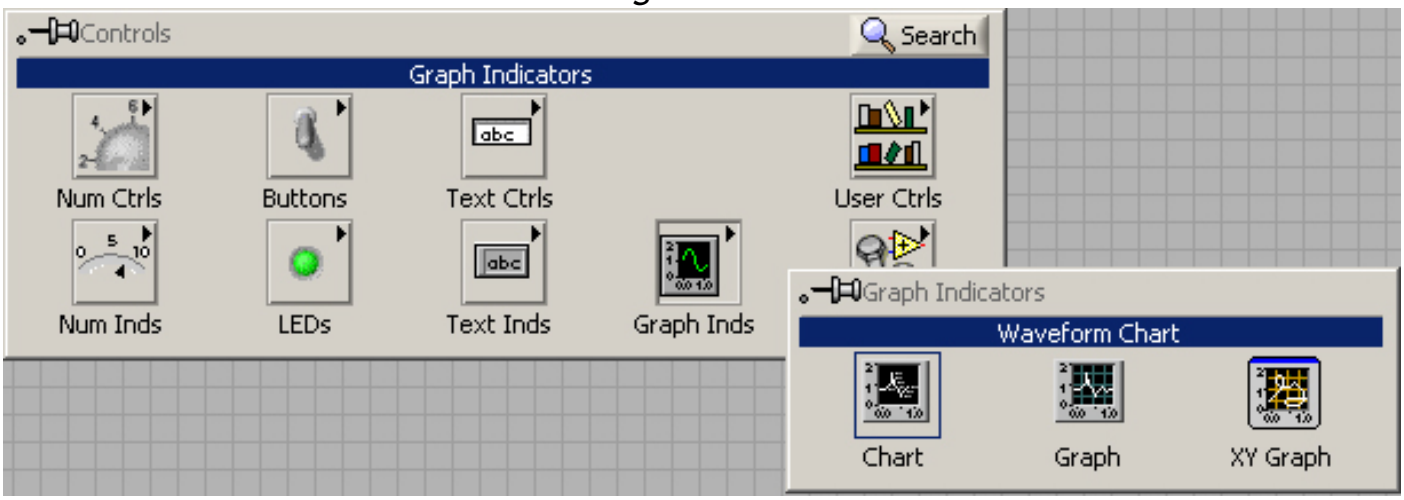


Warning An easy mistake to make when using feedback nodes is to forget to wire the initializer terminal. If you don't wire it, it will pull whatever value it last reported when the program last ran. Also, what it does reports when the program first starts (when it doesn't really have any idea what to report) is somewhat unpredictable in that its dependant upon many things. In general it is best practice to wire the initializer terminal; do it unless you have a specific reason to not do it.

As you may be able to guess from the code, this example does the same exact thing as the example pictured in figure 1.3.1, except it starts at 0 instead of 1.

Now that you know about the feedback node, the last thing that we need to discuss, in order to fully interpret the example demonstrated in Figure 1.3.3, is the Waveform Chart. A Waveform Chart is an indicator that you may place on the front panel by right clicking on the front panel, selecting "Graph Inds," and then selecting "Chart," like so:

Figure 1.3.5



The corresponding indicator icon appears in the block diagram, as you can see above in Figure 1.3.3. Whenever the Waveform Chart is passed data, by default, it scoots over one x value and then plots the new value on the y-axis. This default action can be changed to plot the new value vs. absolute time or relative time as well. You change this behavior by right clicking the chart, selecting properties, selecting the "Format and Precision" tab, and then selecting absolute time or relative time.

Now to introduce the final structure that we will be discussing: the case structure. This structure fulfills the role of both If...Then statements and Case statements in your typical text-based programming languages. Basically, for the If...Then statement functionality, you wire a boolean value to the control terminal of the case structure that controls which part of the case structure will be executed; you only have two options in this case. If you want to have more than one possibility, you can create more than one case, and wire another data type to the control terminal. Here is an example of a simple case structure.

Figure 1.3.6a

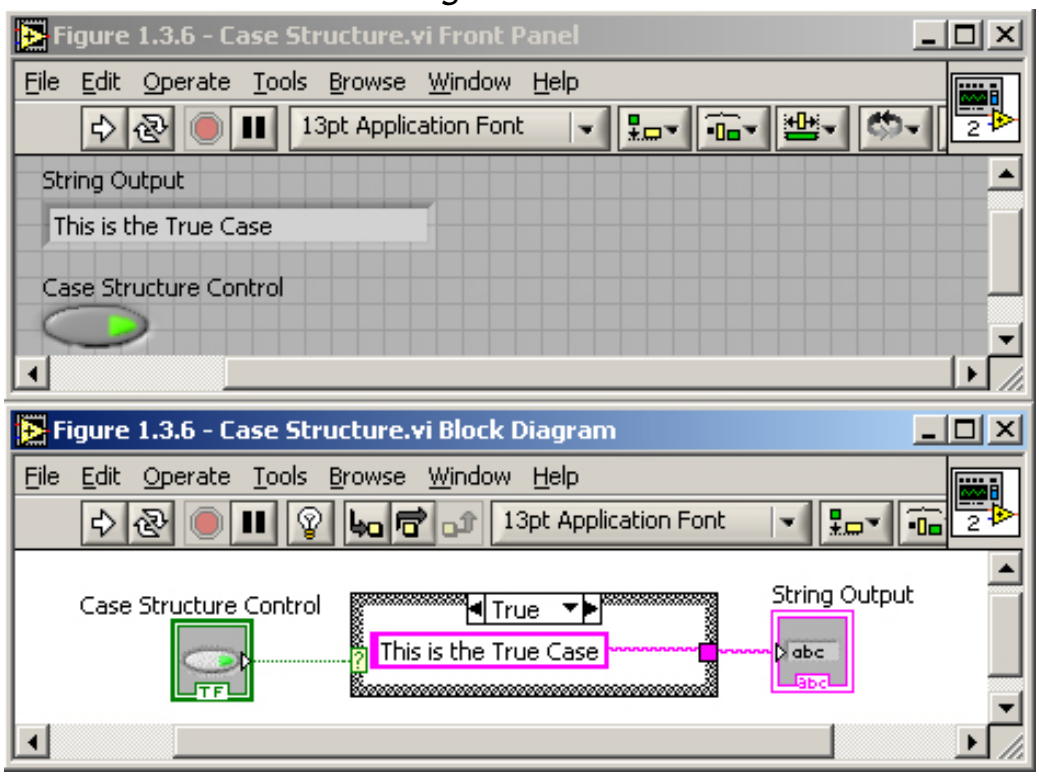


Figure 1.3.6b

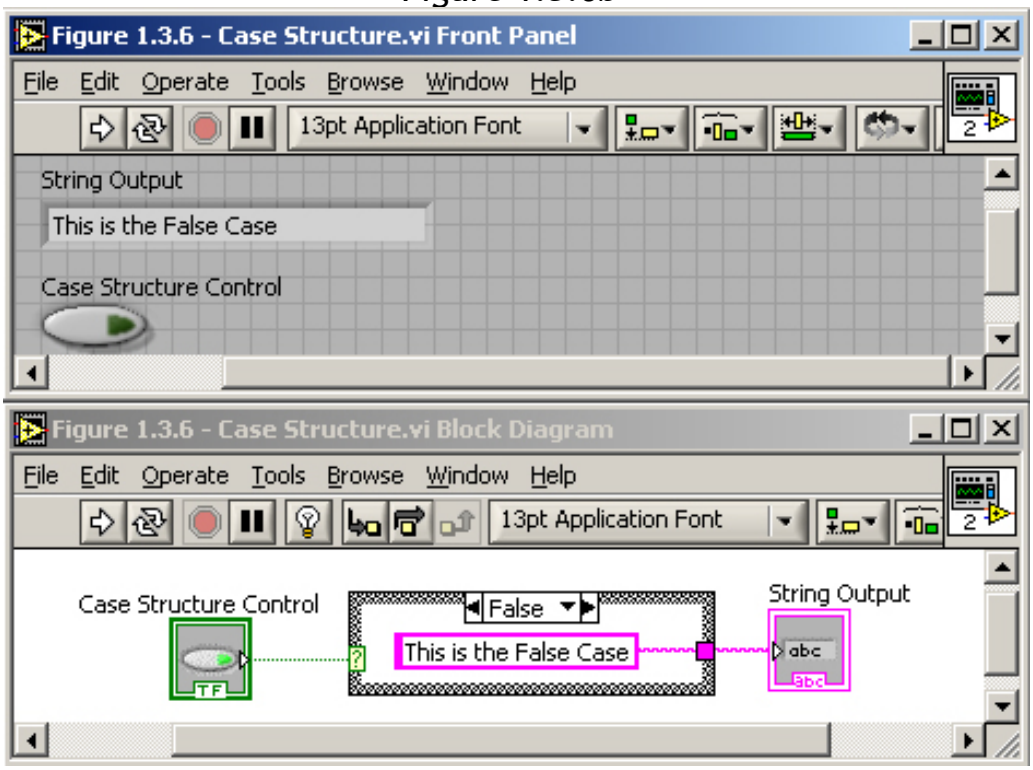
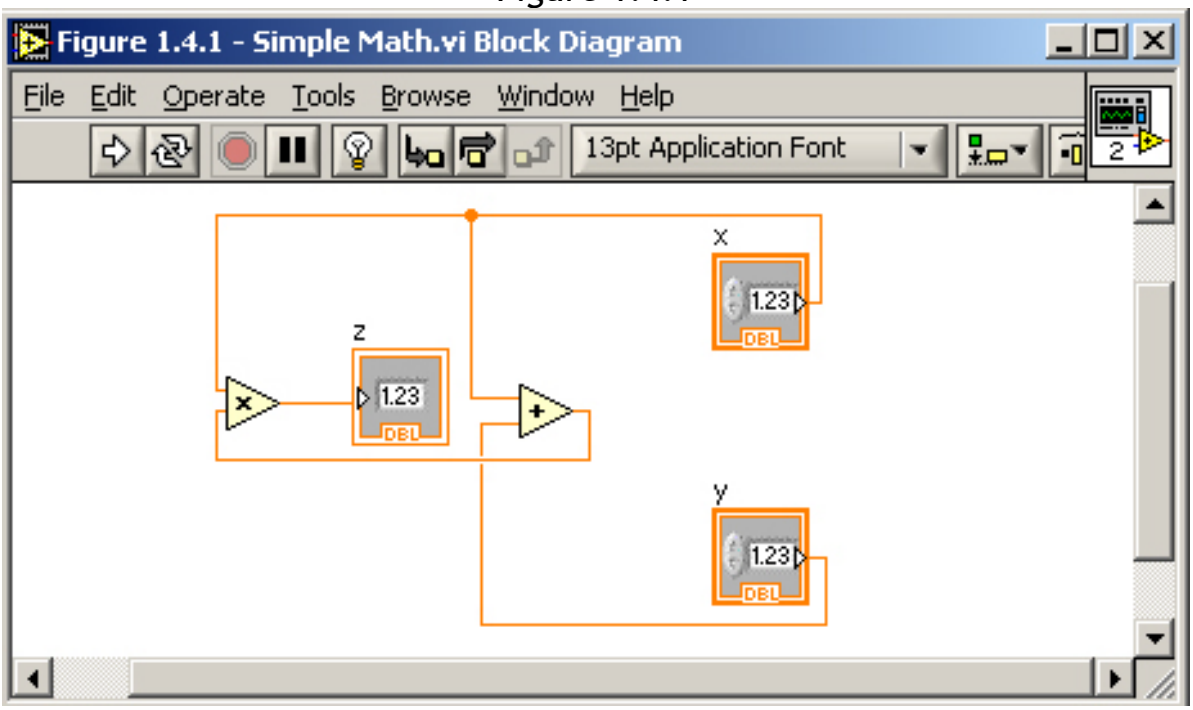


Figure 1.3.6a shows the true case and 1.3.6b shows the false case. As you can see, you put whatever code that you want to execute when the ? box has a true value wired to it in the true case structure box and vice versa for the false case (you change the case by using the arrows at the top of the case structure). You do a similar thing when you want more than one case, but I will leave that up to you to tinker with if it becomes useful to you.

Section 1.4: Data and Execution Flow

The way that LabVIEW handles data flow and order of execution is unique when contrasted with other text-based programming languages. In text-based languages, every command is executed sequentially, and the only way to perform tasks in varying order is to make functions or subroutines out of them. In LabVIEW, the order on the block diagram is irrelevant (although it is somewhat accepted to make programs go from left to right, in general). Take a look at the following example.

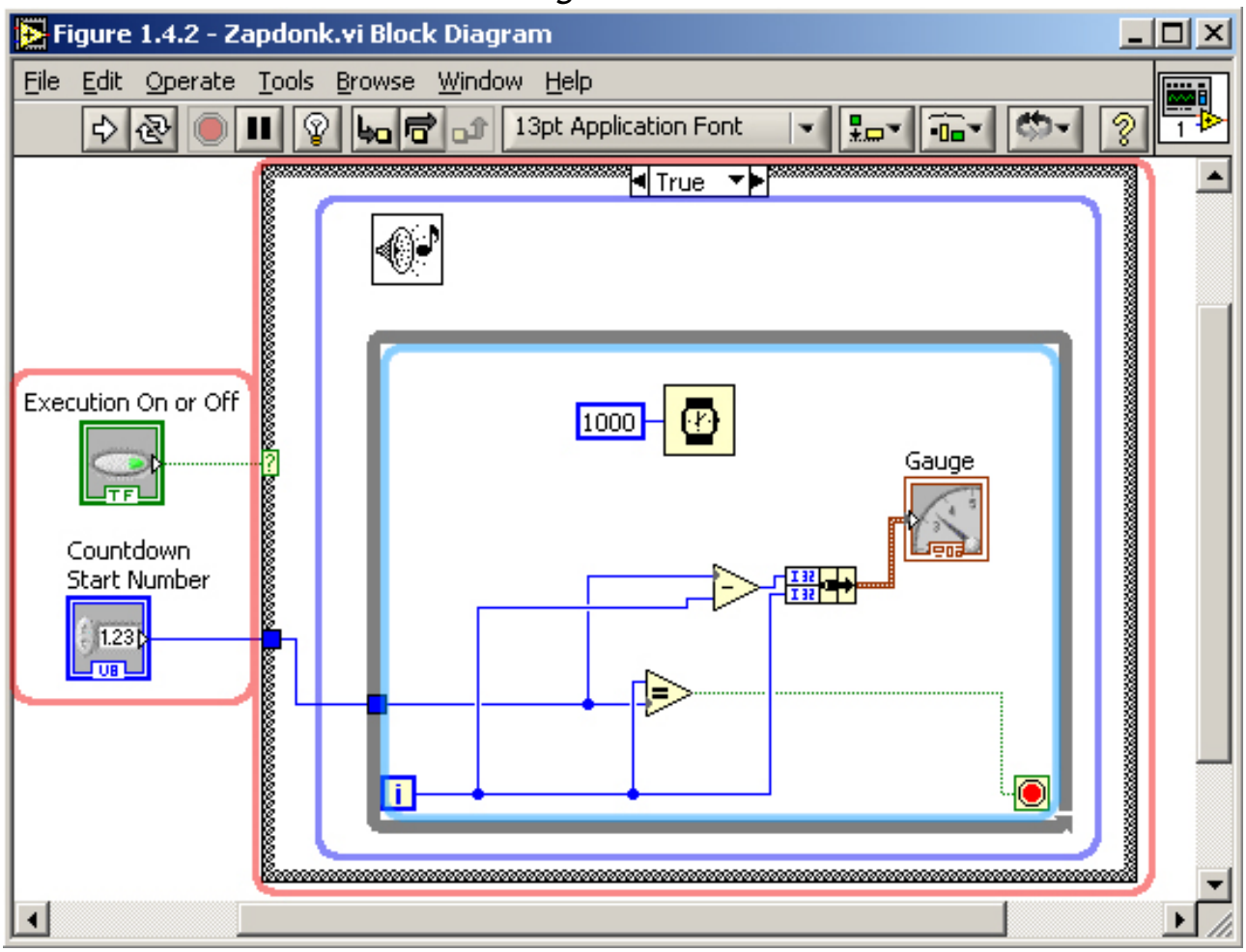
Figure 1.4.1



This program does the same exact thing that the example in Figure 1.2.4 did, except the order is rearranged.

LabVIEW decides execution order based upon the order in which each terminal receives data. It will now be useful to define a term, "zapdonk," to simplify later explanations. This is not a LabVIEW term, but rather one I have coined to better explain this concept. A zapdonk is a collection of commands that has no conditions on its execution at some given part of the execution of the program. This means that once you reach a particular zapdonk in a program, all members of that zapdonk must execute before the program leaves the zapdonk to continue onward to the next zapdonk. As confusing as this is, it deserves an example.

Figure 1.4.2



In this figure, each separate zapdonk is surrounded by a separate color. In LabVIEW, each member of an executing zapdonk that can report data immediately does so. Any member that does not yet have each data terminal supplied with data yet waits until every data terminal is supplied with data before it executes. If it is not supplied with data, it will not execute, therefore it crashes. This is a similar crash to trying to reference a variable before it has been assigned a value in a text-based language. Here is a breakdown of what happens in this example.

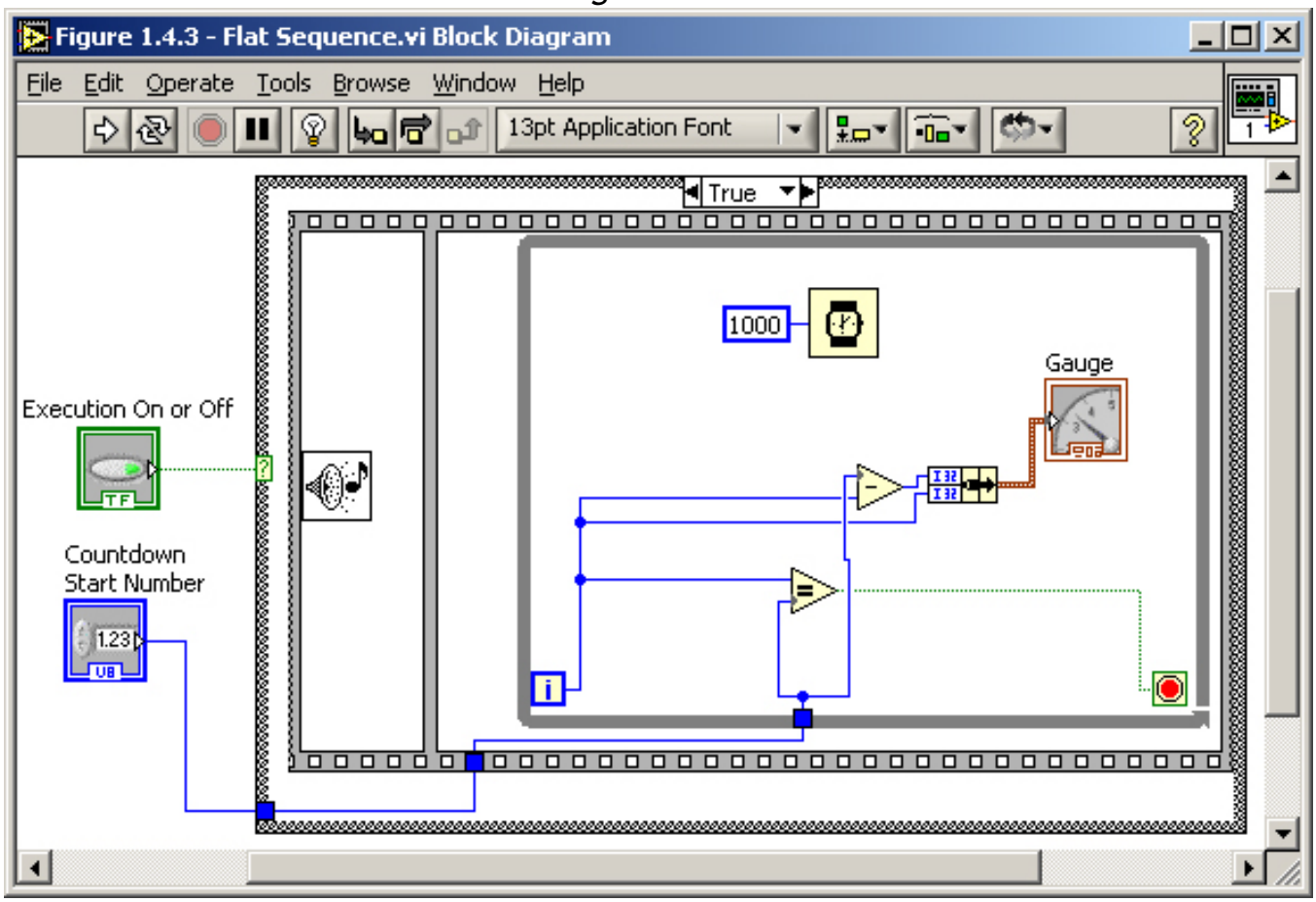
1)Red Zapdonk executes
2)Boolean and numeric-integer control report their data
3)Case commits to perform the true or false statements
4)If true, blue Zapdonk executes
5)The beep subVI executes at the same time (separate threads) that the while loop executes
6)When the while loop executes, the blue Zapdonk executes
7)The "1000" constant/wait path and the counting path both execute at the same time
8)When the loop finishes, the blue Zapdonk will be finished
9)When the blue Zapdonk finishes, the red Zapdonk will be finished
10)The program terminates

This is much easier to see if you go download the example, turn on highlight execution mode,

and watch it execute to see that order that it does things in. The most important thing to remember is that a LabVIEW program executes in the order in which its components are wired, and data is reported "on demand." Considering the fact that all members of a zapdonk execute simultaneously, it can sometimes be a problem to get things to execute in the order that you desire.

To help out with this problem, LabVIEW gives you two tools: the flat sequence and the stacked sequence. Both of these do essentially the same thing, they simply look different on the block diagram. You find both of them under "Structures" on the "All Functions" palette (the same place that for loops and while loops reside).

Figure 1.4.3



This example does the same thing as the example shown in Figure 1.4.2, except it forces the program to execute the beep subVI prior to executing the counting code. A flat sequence executes from the left to the right.

To use a flat sequence structure, simply select it from the Structures area of the functions palette and drag a box somewhere on the block diagram to create the structure. When it is created, simply right click on the border to add frames. After all the frames that you desire are created, add the code to the sequence in the order that you want it to execute, and you're done.

In general, you will want to try to use the wires to control the data flow because it makes your programs much simpler. However, flat sequences are useful in many cases, especially when you need to synchronize different aspects of your program; just make sure you resist the urge to overuse them

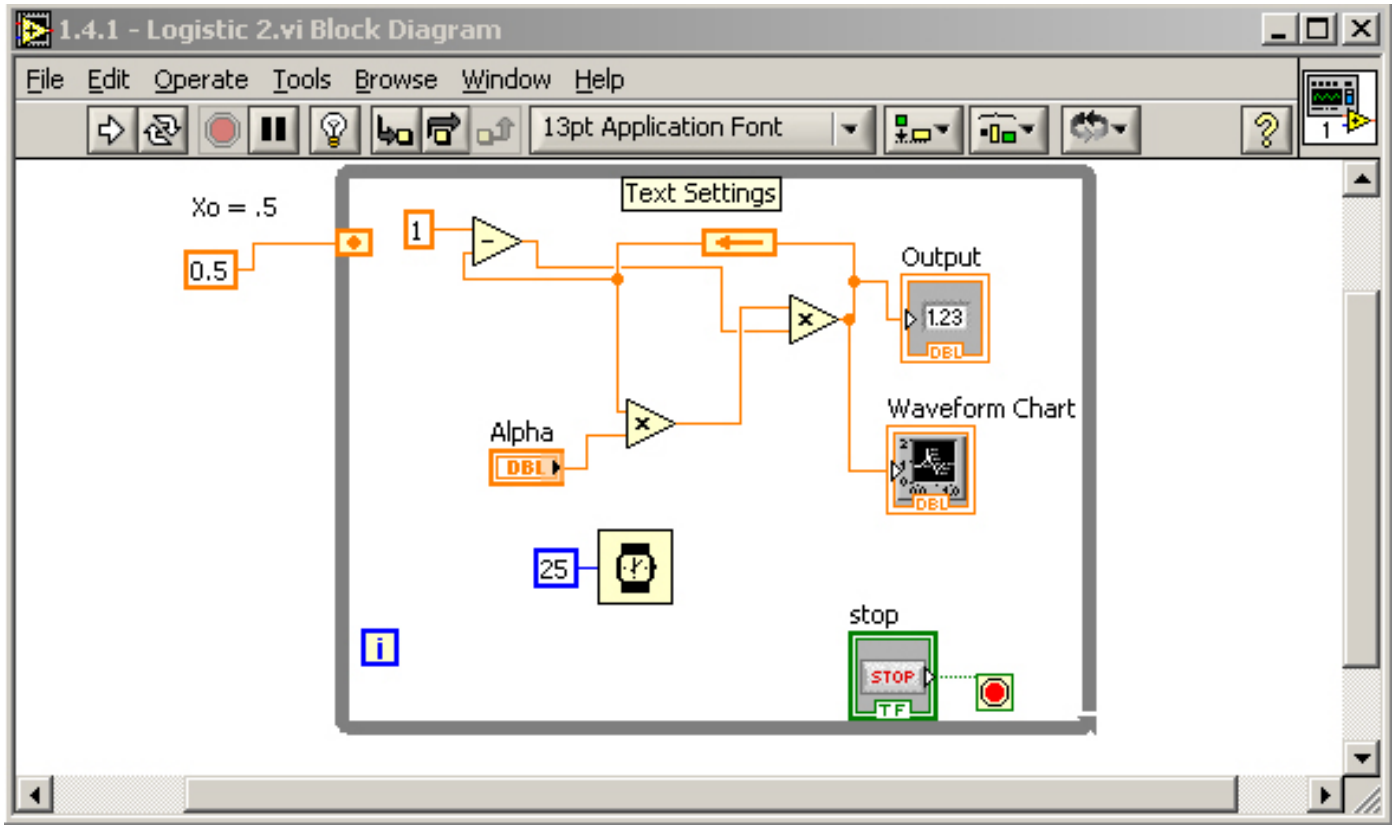
Section 1.5: Interactive LabVIEW Programs

Thus far, the only programs that we have seen have been programs that you simply run, watch it finish, and look at the result. Our goal in this section is to establish a program that will dynamically adapt to user input.

The main component in such a program is, without a doubt, the while loop. In general, you create a while loop around your entire program that has the termination condition wired to some sort of stop button. Let's return to our example regarding the logistic map.

Here is a version of the program that lets you vary "a" as the program runs:

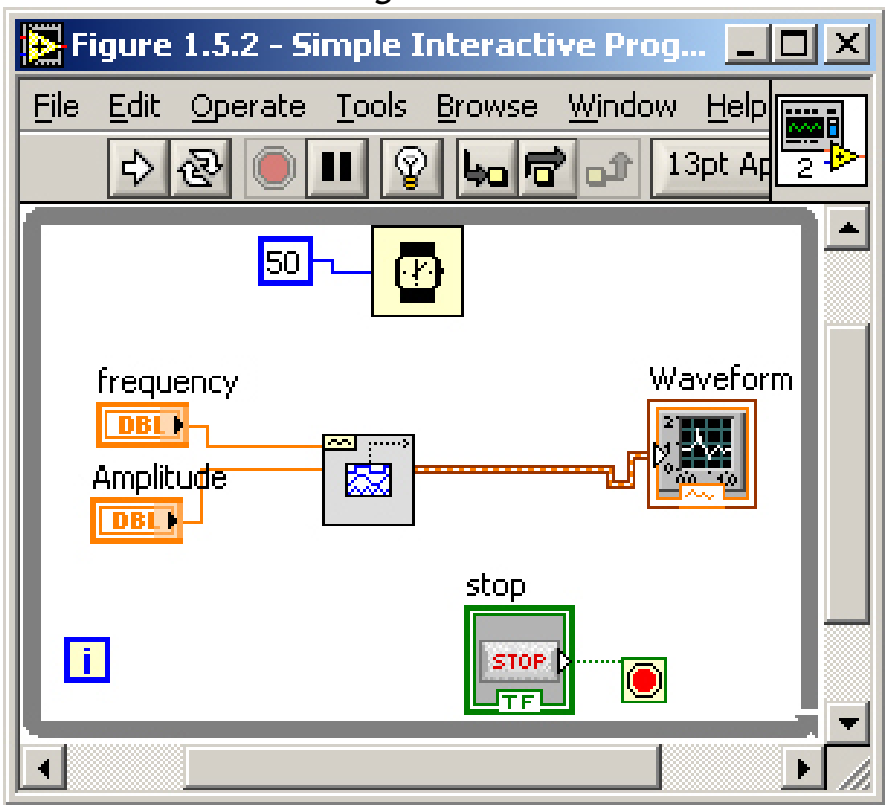
Figure 1.5.1



This program continuously iterates the logistic map and displays the results in real-time on a waveform chart. As you can see from the code, in general, all that you need to do to make a program interactive is assign a control to some parameter and put the entire program in a while loop. This allows the program to continue until a user tells it to stop.

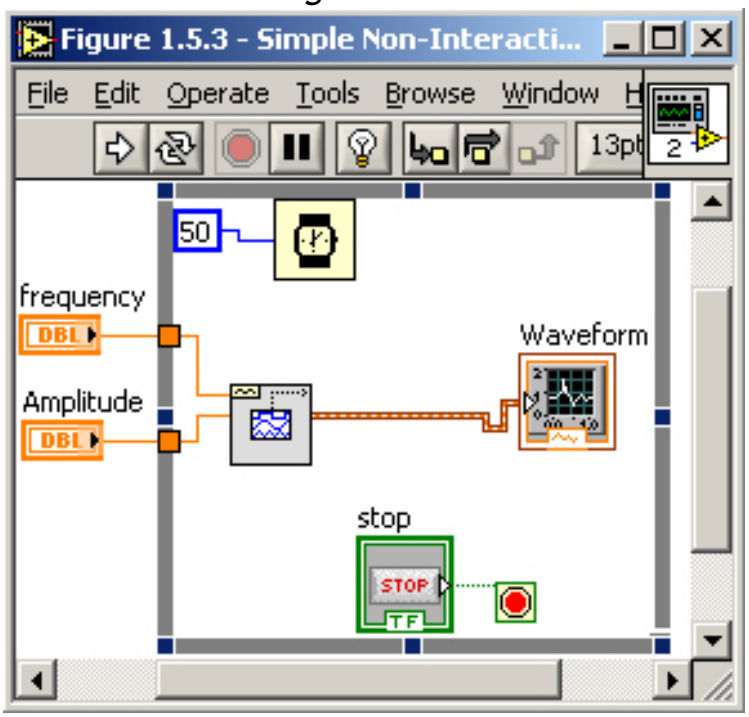
Thus all one needs to do to make a program interactive in LabVIEW is insert a control in whatever loop contains the contains the continuously executing code that you want to control. It is important that you insert the control in the loop and not outside the loop (remember what was said in Section 1.4 about execution order). To demonstrate this, the following example allows for run-time change of parameters.

Figure 1.5.2



And the next example will have the same functionality, except one cannot change the parameters while the program is running.

Figure 1.5.2



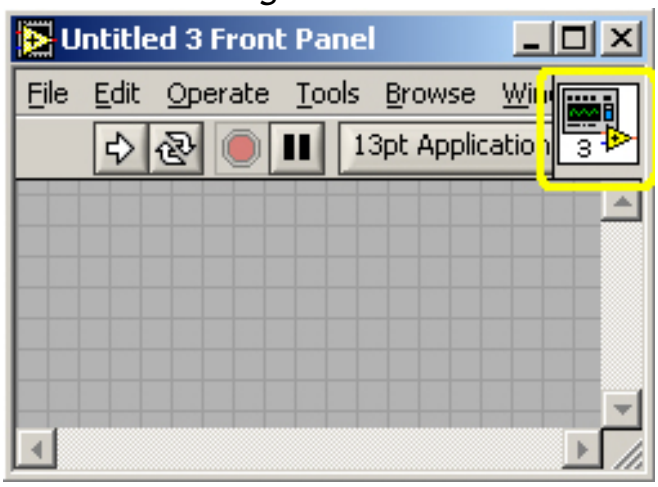
While writing a LabVIEW program, it is generally best to put the control in the same zapdonk as the parameter that it is controlling, in order to provide the user with control over that parameter during the execution of the program. However, there will be many cases in which you will not want the user to control parameters of certain things during execution time (when controlling parameters of a data acquisition device that is actively taking data during the main program execution, for example). Just make sure that you think about exactly what you wish to accomplish with a control when you place it in the block diagram.

Section 1.6: SubVI's

Thus far, we have focused on using the various subVI's that are packaged with LabVIEW. Now we shall learn to make our own. SubVI's are analogous to subroutines or functions in text-based programming languages. There are many reasons to use custom-written subVI's in your programs. For one, you can imagine that as programs get more and more complicated, you tend to run out of room on the block diagram if you put every single bit of code directly in the block diagram you are working on. SubVI's work to organize your block diagram. Secondly, if you work hard to write a good, adaptable subVI you may reuse your code at a later time to do a similar task.

You create a subVI very close to the same way that you create a top-level VI, only you do a little bit extra when you are done working. You begin by creating controls and indicators for every input and output that you want your subVI to have. Then, you use the connector pane to link terminals of your subVI to the corresponding control or indicator. The figure below shows you where the connector pane is located.

Figure 1.6.1



Until you tell LabVIEW otherwise, the connector pane shows the icon for your VI. To show the connector, navigate to the front pane, right click anywhere in the connector pane, and select "show connector." You will see a series of boxes appear in the connector pane. Each of these boxes corresponds to a potential connection that can be made to your subVI. If you right-click somewhere on the connector pane when the connector is showing, you may add/remove terminals and control which connections are required (or optional...called "recommended" in LabVIEW) for execution. Required connections will throw an error when there is nothing wired to it when it is called.

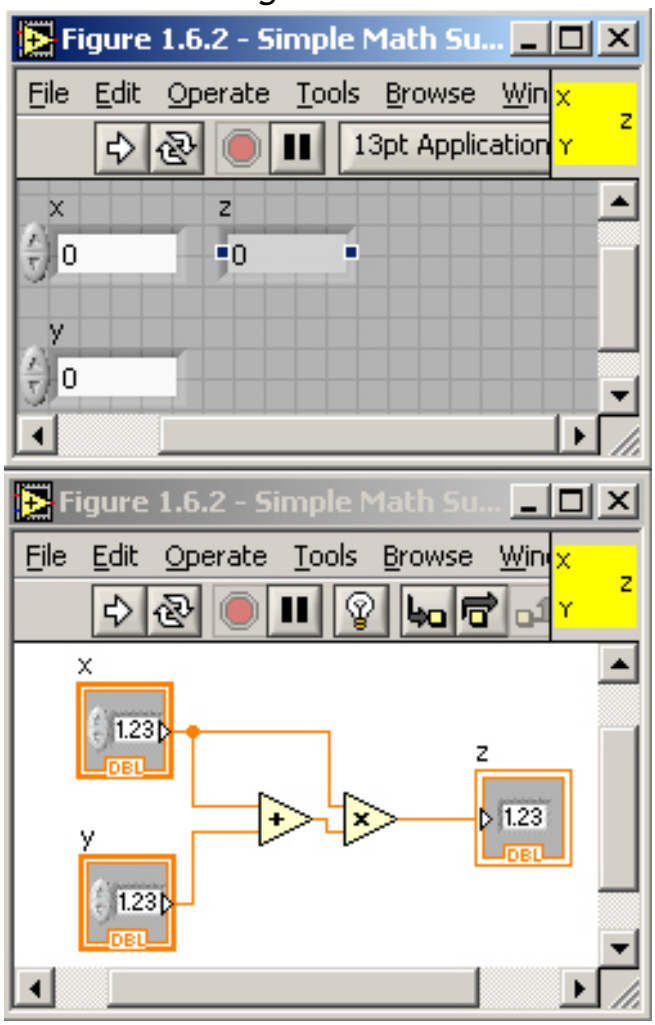
To wire a control or indicator to your connector simply click on a connector box (you'll see the wire spool cursor pop up) and then click on the corresponding control or indicator. You will see the color of the appropriate box in the connector pane change to the color that corresponds to the data type of the control or indicator that you wired to the connector.

Now to design the icon. It is generally best to design an icon in a separate graphics editor such as Photoshop or GIMP. All you must do is design a 32 x 32 pixel graphic that you want to use as the icon and then drag it from anywhere on your file system to the connector pane. However, if you do not have such software available to you, LabVIEW provides a very rudimentary and not very easy to use icon editor.

To access the LabVIEW editor, all you must do is right-click in the connector pane and select “edit Icon.” The icon editor is pretty obvious in its frustrating usage, so I’ll just let you play around with using it.

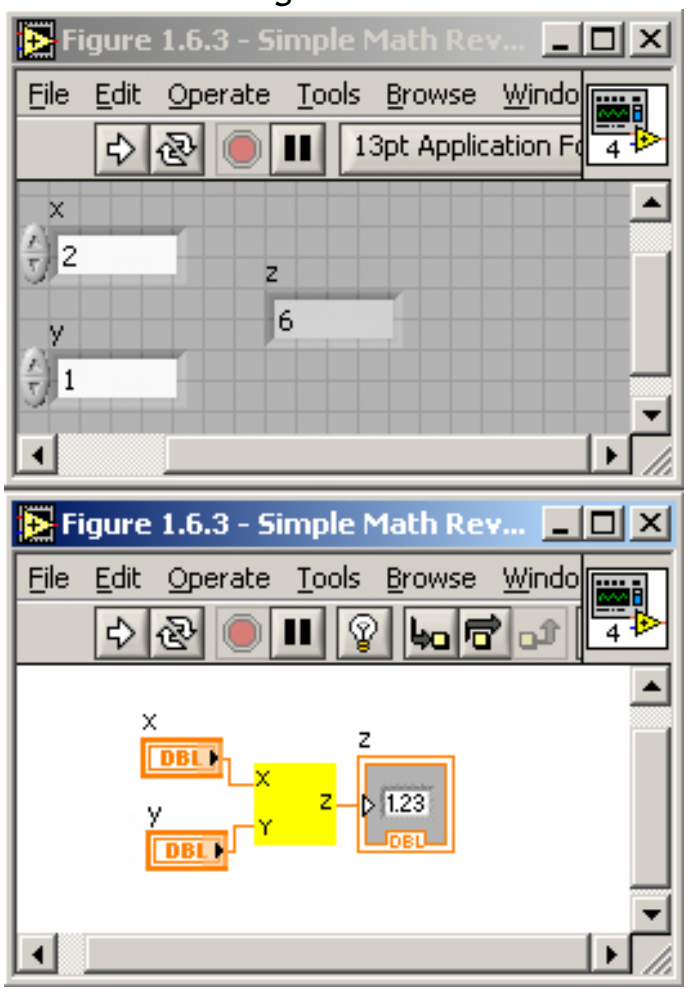
You now know all the basics for creating a subVI; here is an example. I’ll take the first complete subVI that we made and turn it into a subVI (see Figure 1.2.4).

Figure 1.6.2



As you can see, the only differences between this program and the program of Figure 1.2.4 are that is has an icon that I made and that, although you can’t see it, I connected the X, Y, and Z terminals to the appropriate connector in the connector pane. To place a subVI that you have made on the block diagram, you may do one of two things. You may find it in the file system and click and drag it onto the block diagram, or you may go the “All Functions” area of the functions palette, find “Select a VI...” and then navigate to the location of your subVI that you desire. Either way the effect is the same; you end up with your subVI on the block diagram. You may choose which method you prefer. Now here is an example that uses this new subVI from Figure 1.6.2.

Figure 1.6.3



To edit a subVI that you have created when it is already in your block diagram, simply double-click it to open up a new LabVIEW window with the content of the subVI present to be edited. Any changes that you make to the subVI will be reflected in any other VI's that call it, so be careful when you do this.

If you run this program, you will see that it has the same output as the program in Figure 1.2.4; it is simply a little bit neater. While making a subVI to do something this simple is overkill, it is easy to imagine times that the content of a subVI would not be quite so trivial (just double click the Function Generator subVI that we used in Figure 1.5.2 for a prime example of a non-trivial subVI).

There is one other way to create subVI's that is useful when you have already written a program, and you would like to clean it up a bit. To create a subVI this new way, you simply highlight (marquee around) all the code that you have written that you wish to turn into a subVI and then go to Tools > Create SubVI. To demonstrate this new functionality, I will return to the VI found in Figure 1.5.2. All that I will do is select the "50" constant, the Wait subVI, and the function generator then navigate to Edit > Create SubVI. What results is a subVI that contains all the code that I selected; LabVIEW even wires all the terminals to the proper terminals for you.

Figure 1.6.4

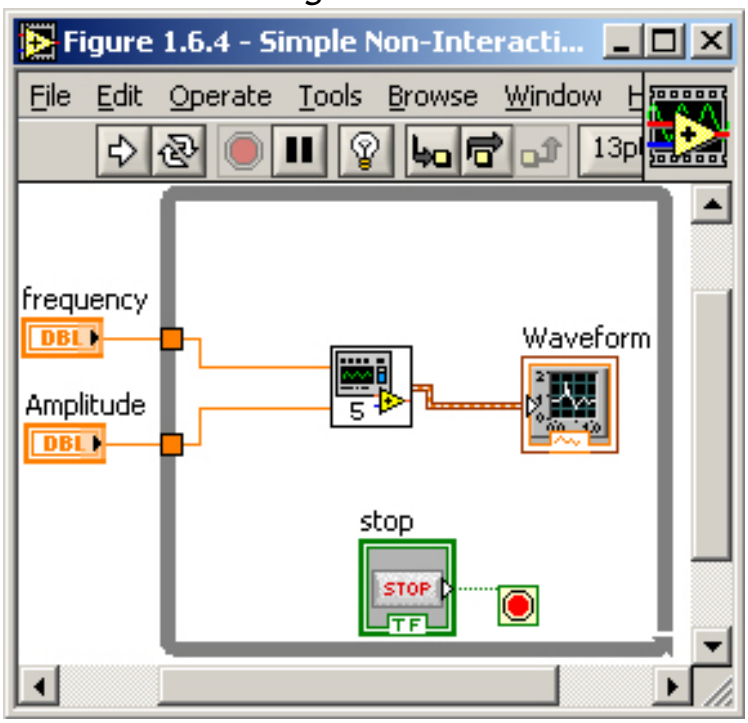
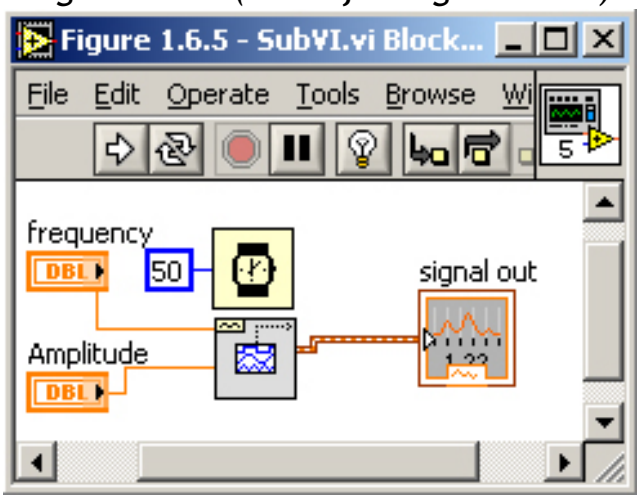


Figure 1.6.4 (subVI for Figure 1.6.5)



As you can see, the subVI simply takes the place of the previous code. The default icon is created for your new subVI; you may double-click on it and edit the icon to be whatever you would like, just like any other subVI.

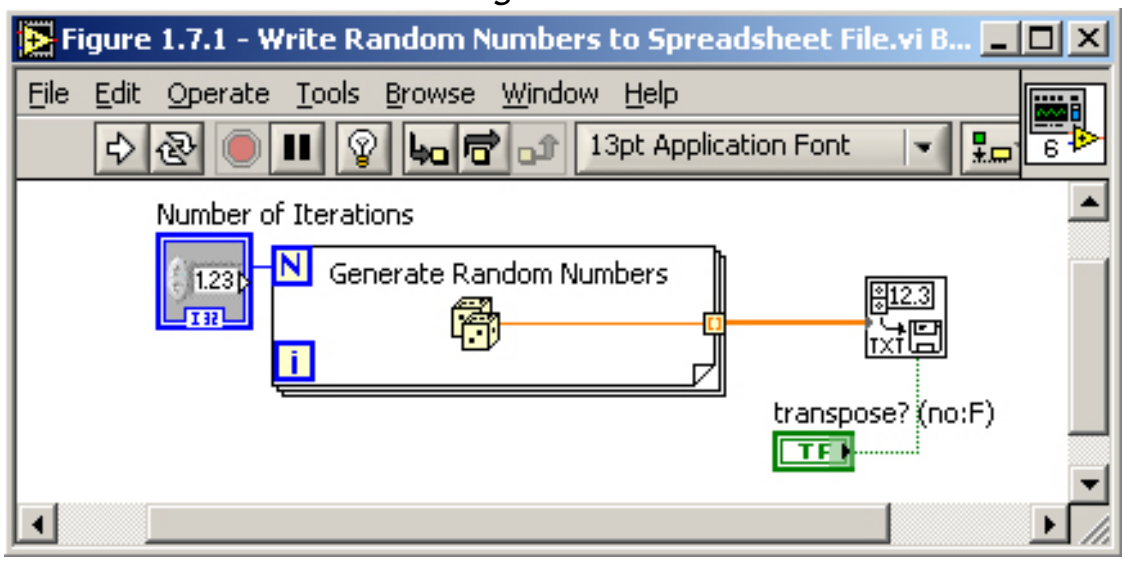
Just remember that when you make a subVI, you remove any potential interactivity with the contents of that subVI--you simply pass it arguments, and it gives you results. Thus you don't generally want to put one of your loops that are designed to provide interactivity (like we discussed in section 1.5) in a subVI, since that removes any control that the user may have over the contents of that loop at runtime.

Section 1.7: Saving Data

There are many ways to save the data that you take in LabVIEW. The primary way that we will focus upon is exporting the data in a spreadsheet format. Luckily, LabVIEW comes with a built-in subVI that handles exporting data in various delimited formats, including comma separated and tab separated (the two that we will be dealing with most often).

Most of the File I/O operations may be found in the File I/O area of the All Functions palette. The main VI that we will be using is the "Write To Spreadsheet File" VI. This VI writes a 1 dimensional or 2 dimensional array to a spreadsheet formatted file using whatever delimiter that you tell it to use. By default, the Write to Spreadsheet File VI uses a tab as its delimiter which is exactly what we want it to use. First, to get a basic understanding of what the VI does, I suggest turning on context help and reading about what it does. When you have a general understanding of how it works, look at the following example.

Figure 1.7.1



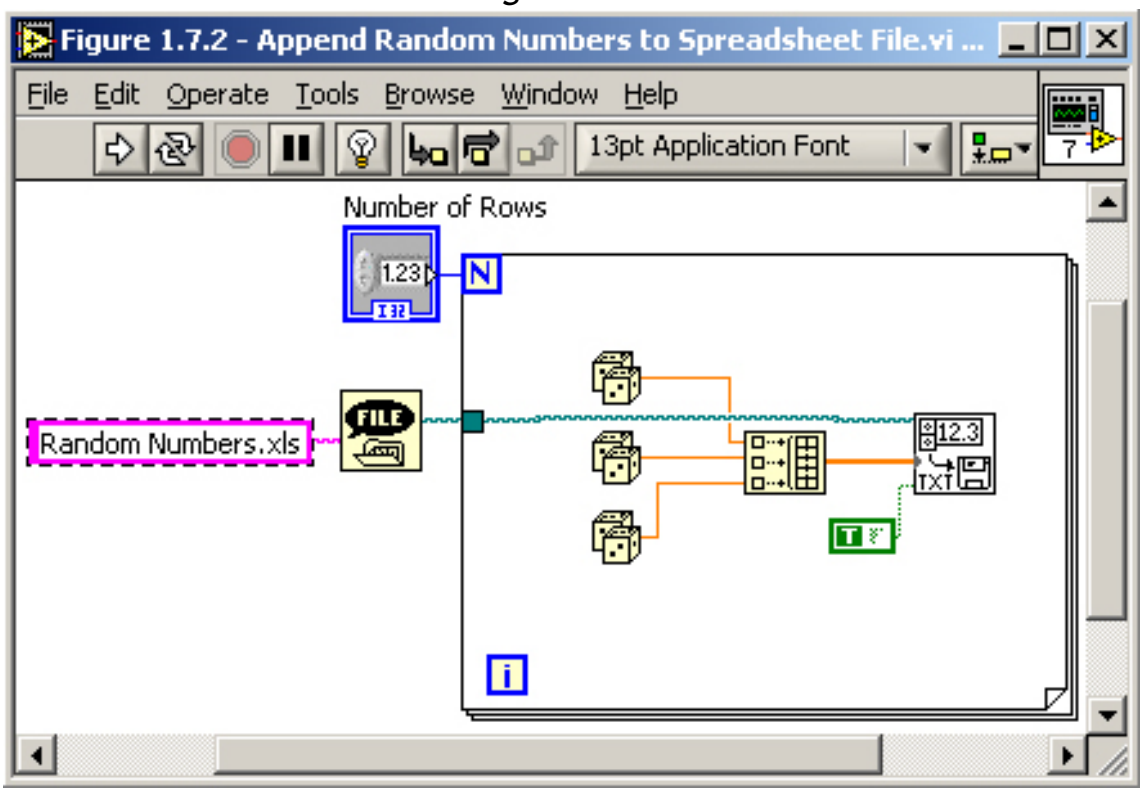
This VI generates N random numbers (the user can control N) and then writes all of them, in the order that they were generated, to a spreadsheet file that is named via a dialog box. The first thing to explain in this example is the indexing role of the tunnel (a tunnel is the little box on the edge of the for loop that gives you access to the data inside of it). Whenever you have indexing turned on (toggled on or off by right clicking on the tunnel and selecting "enable indexing" or "disable indexing"), every iteration of the loop writes appends a new value to an array that grows as long as the loop executes. The net effect of indexing an output is that the 0th element of the output array is the first value that the loop reported, the 1st element is the second value that the loop reported, etc. If you disable indexing, only the final value that is reported as the loop terminates is output through the tunnel.

In this example, we wired the indexed output of the random number generator to the 1D input of the Write to Spreadsheet File VI. This causes the Write to Spreadsheet File VI to write that entire array horizontally if the transpose flag is false or vertically if the transpose flag is true. It is noteworthy that this particular VI only writes numeric data--no strings, waveforms, etc are allowed.

The method described in the previous example is a good way to write data in some cases, but it is generally not the best way to write data that has been acquired in a lab setting. The reason for this is that it doesn't include any sort of disaster recovery, meaning that if for some reason the for loop crashed, all the data that was taken in previous iterations is lost. This is not a big problem for our little random number generation, but it is a huge problem if you have been taking data for a day and a piece of equipment gives you an error that you weren't prepared for that crashes your program (this has happened to me 3-4 times before I changed my way of doing things). Thus we want a way that exports data continuously, not as one big bundle at the very end of execution. We can do this using the append functionality of the Write to Spreadsheet File VI.

Take a look at the following example:

Figure 1.7.2



This example gets a filename/path from the user when the VI is first executed. This filename/path is then passed to a tunnel on the loop. Thus every iteration, 3 random numbers are generated (which, since transpose is not on, means that one row with three columns of data is generated) and appended to the file that is specified by the filename that was acquired at execution. On the first iteration, when the file doesn't exist, the file is created.

The major concern with this method is finding some way to make sure that the user does not append data to a file that already exists, unless they specifically mean to. In most cases, you will want to include functionality that keeps users from making this sort of mistake by throwing an error if the file already exists when you first procure the path and filename. Unfortunately, this is a relatively complicated process that is out of the scope of this particular paper; ask me how if you would like to know, and I'd be happy to tell you.

Section 1.8: Data Acquisition

One of the strengths of LabVIEW lies in its seamless integration of data acquisition. The process is simplified even more if you use National Instruments hardware, since they have absolutely superb drivers written for all of their equipment. When you are using NI hardware, you may use a handy VI called the DAQ Assistant, which is found under the "NI Measurements" area of the All Functions Palette. This VI allows you to create virtual channels that serve as a handy, migratable way of handling data acquisition. First, however, I must introduce you to MAX (Measurement Automation eXporer). Here is a screen shot of MAX that also shows where the USB-6008 appears.

Figure 1.8.1

If you right click on a device, you will see a selection entitled "Test Panels." The test panel allows you to do simple read/write operations for testing purposes. You may also rename a particular device from the default of "Dev1" to something more user-friendly. This device name is what you will refer to when you are using LabVIEW to interface with the device.

Getting data from a NI data acquisition device is fairly simple. The first step is to put a DAQ Assistant VI on your block diagram. You may find this VI under the "All Functions" in the "NI Measurements" section. When you first place the VI on the block diagram, it pops up a dialog box that lets you configure your data acquisition job. Here you choose what you want to measure (we will be using voltage analog input), what channels you wish to use, what acquisition mode you wish to use (continuous, N samples, etc). After you finish, you may reconfigure the VI by double clicking the DAQ Assistant VI.

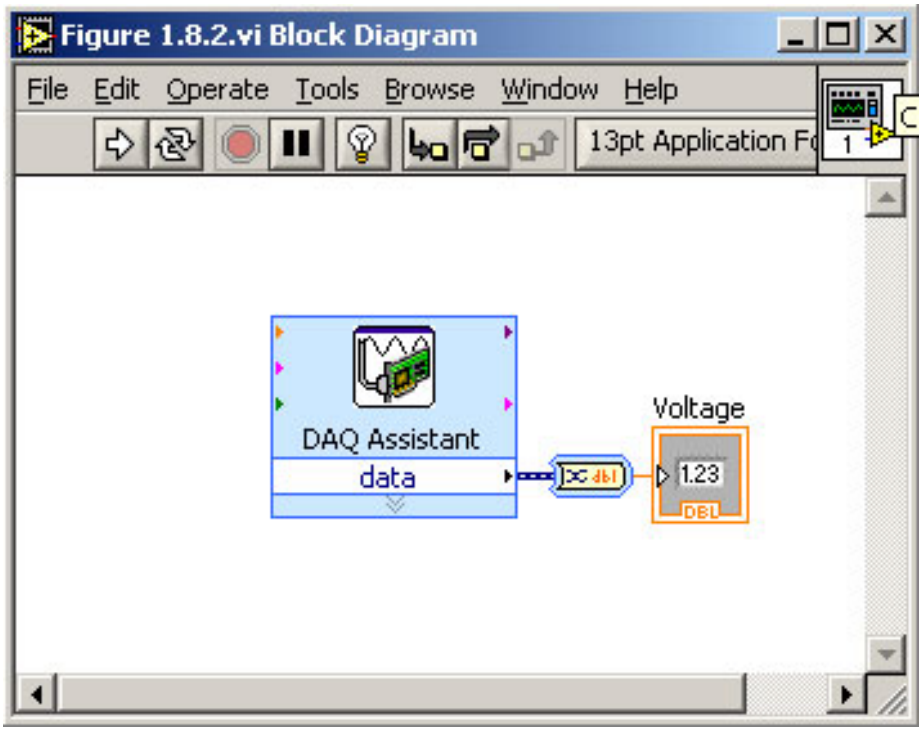
The DAQ Assistant VI outputs data in a data type called "Dynamic Data." This data type is basically a way of keeping all the channel information, number of samples, acquired data, device information, etc., all in one place. Often times, you must convert this data type to other data types to make use of the data. For example, if you take only one sample and you just want a number to represent the measurement (commonly a voltage) you must extract the sample value and convert that to a scalar. You use the "Convert From Dynamic Data" VI to do that.

The "Convert From Dynamic Data" VI is found in the "Signal Manipulation" area of the de-

fault Functions popup, next to the “All Functions” area (in other words, it is in the first area that pops up when you right click on the block diagram). Once you place it on the block diagram, it will pop up a dialog box that allows you to choose what output format you wish to use. Generally, the simples and most common use is the “Single Scalar” output format, but you may choose whatever is most useful to you. After you click “Ok,” all you must do is wire the dynamic data source to the VI, and then wire the output to whatever you wish.

Thus, in general, there are two main steps to performing data acquisition: configure the DAQ Assistant and convert the dynamic data to a more specific format. Here is an example that does just that; as you can see, it takes very little code to perform a data acquisition task.

Figure 1.8.2



This VI Simply reads channel zero (as configured by the dialog box associated with the DAQ Assistant) and outputs its data to an indicator. When you are using NI-DAQmx-ready hardware (such as the USB-6008 that we are using), it is as simple as telling the DAQ Assistant what you want to do and then letting it do the rest.

Section 1.9: Closing Comments

Now you know the most important basics of programming in LabVIEW. There are many, many aspects of LabVIEW that I have not covered here, but I believe that after you have read this, you will understand enough to be able to go out on your own, read some documentation, and understand what you read.

When you finally get to the lab exercise, you will undoubtedly run into something that you need to do that you do not know how to do. I encourage you to first familiarize yourself with the help menu (there are many nice examples, as well as very thorough documentation available to you) and then with my e-mail address, mhollin3@utk.edu. I will be happy to help in any way possible.

Furthermore, if you want more information, there are some very good forums available to LabVIEW users to be found at http://zone.ni.com/devzone/cda/main. There are many people there who know what they are talking about, and chances are, whatever your problem is, it has already been solved; the problem lies in tracking down the answer to it.

Finally, over time I will have more and more LabVIEW content available on my web site, http://www.evanescenthorizons.com/. For now, I have all the examples contained in this text available online under Labs > Introduction to Modern Data Acquisition. That is also the location of the electronic version of this text, assuming at the moment that you are reading a printed version...otherwise, I guess you found it :).

MATLAB

Section 2.1: Introduction to MATLAB

MATLAB is an industry standard numerical computing tool that is useful in all kinds of scenarios. Its prime use that we will be focusing upon is that of data analysis. MATLAB comes with many built in functions that can import data formatted in common spreadsheet formats (such as tab and comma separated), and we will be using these functions to import data for analysis when we finally come to the lab exercise itself. In the meantime, I will work to familiarize you with the basics of the MATLAB platform.

While MATLAB contains very high quality, predictable functions, a large percentage of its strength lies in the sheer number of content that it contains. There are thousands of functions, including ODE/PDE solvers, three dimensional interpolators, data fitting algorithms, numerical and symbolic differentiation/integration, etc. Plus, it comes with a complete and extensive visualization functionality that includes both 3d and 2d graphing capabilities. It is this graphing functionality that will be the main focus of our dealings with MATLAB when it comes time to work on the actual lab exercise.

In addition to performing singular, one shot calculations, one may program MATLAB to do certain tasks or sets of calculations that you wish to use again in the future. "Programming" in MATLAB is more akin to scripting than anything else. In fact, code written in MATLAB is commonly referred to as a MATLAB script or m-file (named for the fact that the typical MATLAB script file extension is .m). Scripting in MATLAB is straightforward once you get used to using the command window, since you simply type the same thing in the script that you would type in the command window. Since you will be introduced to MATLAB first by using the command window, scripting, for the most part, will come along of its own accord.

My example scripts will be in the following format:

Example 2.x.x

```
%Filename
This is some sample MATLAB code
This is some more MATLAB code
Yep you guessed it...more MATLAB code
```

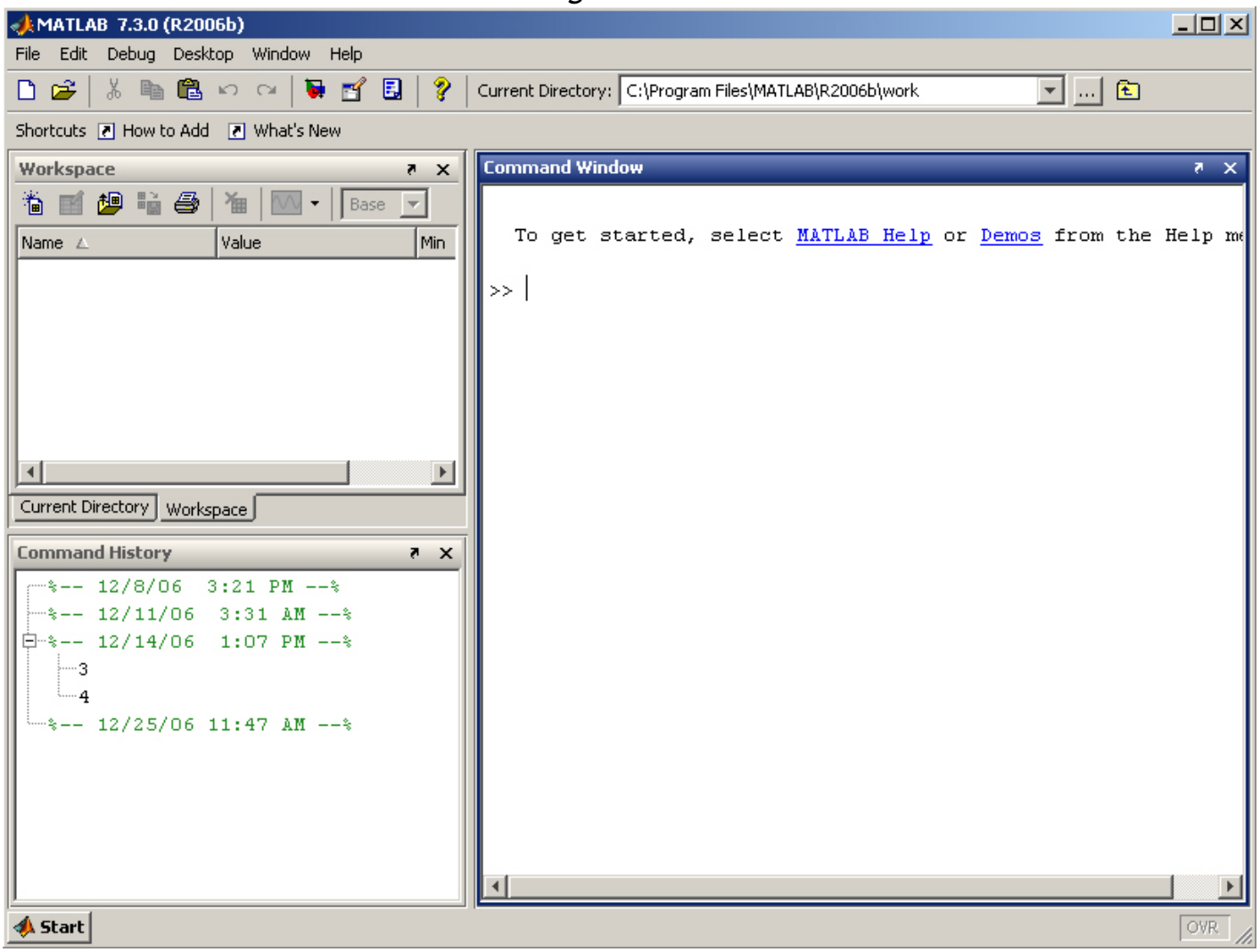
My example code, on the other hand, will simply be in line, like so:

```
>> MATLAB command
|| Output for MATLAB command
>> Second MATLAB command
|| Output for Second MATLAB command
```

In the example code, a return is implied for every new line, in other words, you would type "MATLAB command" then type enter before going on to the second line of commands. The >> denotes input while the || denotes output.

Now that all the introductions are out of the way, I will begin by showing you around the MATLAB interface. When you start up MATLAB, you should see something similar to the following:

Figure 2.1.1



The biggest window, called the "Command Window," is where you do most of your work. It is here that you call functions, do arithmetic, assign variables, call your scripts, etc. The "Workspace" window shows any variables that you have created. It allows you to see what kind of variable it is, check its value, delete it, or edit it. You may also import variables by loading previously saved workspaces (saved as .mat files). The "Current Directory" tab that is underneath the "Workspace" tab is the area that allows you to see the contents of your current working directory. It also lets you change your working directory to another directory if you would like. Your current directory dictates what scripts will be executed when you type their name into the command window. For example if you have a script called "MyScript.m" in your current working directory, you could type `MyScript` into the command window to execute your script.

You may customize your interface by dragging each of these windows around your work area. Simply click on the window title and drag it around to reposition or tab it in another window. If screen real estate is important to you, you can tab all windows together in one area. You may also click the little arrow in the top right-hand corner of a window in order to give that window its own frame in Windows. Additionally, there are more windows that you may find useful; you may find these under the "Desktop" menu.

Section 2.2: Simple Math with MATLAB

Now to do some math in MATLAB. Take the following input to MATLAB for example:

```
>> 2+2
|| ans =
||         4
```

(remember the the || is simply my way of telling you that the text is output... MATLAB does not really output that)

When you input `2+2` into MATLAB, it returns the answer by assigning the answer to a variable called ans. If you look under your workspace window, you will see that a variable named ans has indeed been created. MATLAB always assigns its results to a variable. It can be a variable that you assign, or if you don't explicitly assign a variable, it assigns it to the `ans` variable.

A variable assignment is pretty straightforward in MATLAB. All you have to do is state the variable name, type an equal sign, and then type what you want that variable's assignment to be. Here are some examples

```
>> x = 2
|| x =
||          2
>> y = 2+2
|| y  =
||          4
>> z = x+y
|| z =
            6
```

Make sure you don't accidentally put the variable **after** the equal sign, ie, `2=x`, as that will give an error.

All of these variables are of a numeric data type; however, there are other types of variables. We will discuss others when the time calls for them. Regardless, all variable assignments are made in the same way, i.e., `<variable name> = <desired expression>`.

Also, if you wish to suppress the output of anything in MATLAB, simply put a semi-colon after the expression that you wish to silence. For example, if you would like to set x equal to 4, but you don't want MATLAB to repeat the assignment to you, do the following:

```
>> x = 4;
```

You will see no output when you do this; however, you may see that the variable was properly assigned by looking in your workspace.

Now, onward to functions. MATLAB uses the common and familiar parenthetical method of passing arguments, which is `function_name(argument_1, argument_2, ... argument_n)`. To use a function in MATLAB, you simply type the function's name and arguments (separated by commas) and hit return. For example:

```
>> x = peaks(30);
>> y = sin(pi);
>> z = yourownfunction(2);
```

MATLAB comes with thousands of very useful functions (which is the most powerful part of MATLAB), so we will be using functions quite extensively.

Now that we have the basics out of the way, we will concern ourselves with the next most important syntax consideration in MATLAB: matrices.

Section 2.3: Matrices and Vectors

MATLAB sees all numeric variables as matrices; single numbers are seen as 1 x 1 matrices. A vector, to MATLAB is any 1 x n or n x 1 matrix. There are many ways to go about creating matrices. One of the easiest ways to deal with two dimensional ones is to use the bracket syntax. It looks something like this:

```
>> x = [1,2,3;4,5,6;7,8,9]
||         1     2     3
||         4     5     6
||         7     8     9
```

You start in the upper left-hand corner of the matrix you are creating, and work your way across, left to right. Commas separate members of the same row, while the semi-colon separates the rows themselves. Alternatively, you may assign elements in an array individually by using a parenthetical syntax in the form of `variable(row,column)`, like this:

```
>> x(1,1) = 2
|| x =
||
||      2
>> x(1,2) = 3
|| x =
||
||      2     3
>> x(2,1) = 4
|| x =
||
||      2     3
||      4     0
>>x(2,2) = 5
|| x =
||
||      2     3
||      4     5
```

MATLAB displays zeroes when it doesn't have any data for an element that must be displayed, such as in the third example above. To make it a little more obvious, here is another example:

```
>> x(6,6) = 2; x(3,3) = 1
||
|| x =
||
||      0     0     0     0     0     0
||      0     0     0     0     0     0
||      0     0     1     0     0     0
||      0     0     0     0     0     0
||      0     0     0     0     0     0
||      0     0     0     0     0     2
```

(as you may have already figured out from the last example, you may use semi-colons as sepa-

rators for commands if you wish to type multiple command on one line)

You may also use this syntax to create nth dimensional matrices by the generalized version of this syntax: `variable(dimension1, dimension2, dimension3, ... dimensionN)`. Here is an example:

```
>> y(2,2,2) = 2
|| y(:,:,1) =
||
||     0     0
||     0     0
||
|| y(:,:,2) =
||
||     0     0
||     0     2
```

When you tell a variable to be a multi-dimensional matrix in MATLAB, and you ask MATLAB to display the contents of said matrix, it will display a series of two dimensional matrices that collectively reflect the contents of the variable . You can see this in the above example. So that you fully understand what you are seeing, I will explain what the colons mean.

You may use colons to tell MATLAB to display all of the contents of that matrix dimension. In other words, if you want to display all the rows in one specific column, you type `variablename(:,column_index)`. Similarly, if you want to display all the columns in one row, you type `variablename(row_index,:)`. Here is an example:

```
>> x = [1,2;3,4]
|| x =
||     1     2
||     3     4
>> x(:,1)
|| ans =
||     1
||     3
>> x(:,2)
|| ans =
||     2
||     4
>> x(1,:)
|| ans =
||     1     2
>> x(2,:)
|| ans =
||     3     4
```

You may do numeric operations on matrices. Adding and subtracting matrices adds and subtracts each corresponding element of a matrix. If you wish to do the same for multiplication, division, and exponentiation, you must use the “dot operators.” The dot operators are `.*`, `./`, and `.^`. These to element by element operations on the matrices. Of course, to do any of these operations, you need to have matrices of equal dimensions. Here is an example of each operator:

```
>> a = [1,1;1,1];
```

```
>> b = [1,2;3,4];
>> a+b
|| ans =
||      2      3
||      4      5
>> a-b
|| ans =
||      0     -1
||     -2     -3
>> a.*b
|| ans =
||      1      2
||      3      4
>> a./b
|| ans =
||    1.0000    0.5000
||    0.3333    0.2500
```

You can also generate monotonically increasing matrices very easily using the syntax `matrix=startnumber:stepsize:stopnumber`, like so:

```
>> t=0:1:5
t =
     0     1     2     3     4     5
```

Finally, you may also concatenate matrices. This is a very straightforward process; you simply add the matrices that you want to concatenate together as elements of the bigger, final matrix. Here is an example:

```
>> a = [1;2;3];
>> b = [4;5;6];
>> c = [a,b]
|| c =
||      1      4
||      2      5
||      3      6
>> d = [a;b]
|| d =
||      1
||      2
||      3
||      4
||      5
||      6
```

You may also do everything in one step, like so:

```
>> mat = [[1;2;3],[4;5;6]]
|| mat =
||      1      4
||      2      5
||      3      6
```

That's all you need to know about matrices at the moment; onto M-Files.

Section 2.4: M-Files

M-Files are used for scripting in MATLAB. Now that you have been introduced to the command window, over half the work of understanding the scripting interface in MATLAB is finished. Basically, an M-File is simply a text file that has all of the commands that you wish to execute, separated by either a return/line-feed or by semi-colons. The percent sign (%) designates the area beginning at the percent sign to the end of a line as a comment.

Here is an example of an M-File:

Example 2.4.1

```
%BasicMFile

%This M-File takes a number that you specify, num1, and passes it as an
%argument to the peaks function, and then does a surface plot of the result

num1 = input('Please input desired matrix dimensions: ');

surf(peaks(num1))
```

This example would work exactly the same if you typed all of the non-comment commands into the command window. `surf` is the surface plot function that is built into MATLAB, and `peaks` is a sample three dimensional function that plugs nicely into `surf`. We will have a more detailed discussion of the `surf` function in the next section.

Right now, I want to talk about the `input` function. The `input` function displays a message, that you specify as an argument for the function, in the command window, and waits for the user to input data. When the user inputs data and hits return, the `input` function returns the value that the user put in. In the M-File above, we assigned the user's input to a variable called `num1`, which we then passed to the `peaks` function. The input function simply provides a way to make your M-Files more dynamic.

You can use the input function to provide the familiar "Press any key to continue" functionality in a program also. Simply make the message whatever you like, and ignore the output.

Another useful thing to do in an M-File is to make a for loop. While you can do these in the command window as well, they are much more useful in the context of a script. In MATLAB, the syntax for a for loop is very simple:

```
for iteration_variable=startindex:stepsize:endindex
     <do some stuff>
end
```

Here is a simple example that counts from 1 to 10 for you:

Example 2.4.2

```
%Ex-2.4.2

for i=1:1:10
    i
end
```

This example simply prints the iteration variable once every iteration.

While loops are very similar to for loops as far as syntax goes. The syntax for a while loop is as follows:

```
while <condition>
     <insert commands here>
end
```

Here is an example:

Example 2.4.3

```
%Ex_2_4_3

i = 1;
while i <= 10
    i
    i=i+1;
end
```

This example does the same thing that Example 2.4.2 did, only it does it with a while loop.

Finally, you may loop through different elements of matrices by doing the following:

Example 2.4.4

```
%Ex_2_4_4

a = input('Input any n x n matrix: ');

SIZE = size(a);

for i = 1:SIZE(1)
    a(i,:)
end
```

This example iterates through each row, displaying each one. The `size` function accepts a matrix as an argument, and it returns the size of that matrix as a 1 x n matrix, where n is the number of dimensions of the argument. You can use the `size` function to dynamically loop through matrices of different sizes.

Section 2.5: Visualizing Data

MATLAB offers a great many ways to go about visualizing data. You can do all kinds of basic 2d plots, as well many 3d plots. We will mostly be focusing on 2d Cartesian plots, 3d surface plots, 3d contour plots, and 2d contour plots. Here is a quick reference for the main visualization functions present in MATLAB.

2d Cartesian: `plot`

2d contour: `contour`

3d surface: `surf`

3d contour: `contour3`

Each of these functions have many levels of functionality; we will only be covering the basics. If you want to understand the more fancy things that they can do, check out the MATLAB help file entries for these functions for more detail.

First, we will start with the simplest one of all of these: the `plot` function. The plot function, in its most basic form, simply accepts an x array and y array of the same size and pairs them. If you want to plot a function--say, $f(x) = x^2$, you would need to generate a matrix that has all the x values in it, and then proceed to use the `.^` operator on that matrix to generate the y values. Here is an example that does just that:

Example 2.5.1

```
%Ex_2_5_1

x = -4:.1:4;
y = x.^2;

plot(x,y)
```

If you run this M-File, you will see a nice graph of $f(x) = x^2$ pop up. You can do this with any sort of function you wish. If you want to change the way that the line looks from the default of a solid blue line to, say, a series of red circles with red lines connecting them, you would do the following:

Example 2.5.2

```
%Ex_2_5_2

x = -4:.1:4;
y = x.^2;

plot(x,y,'r-o')
```

This is just one of the many formatting choices that you have; I just wanted to let you know that you have the option of changing how it looks. There are many, many more options. Just look in the documentation for more information on how to go about formatting your graph as you would like it. Contact me if you can't find the information, and I'll point you in the right direction.

That's about it for the `plot` function; onto the 3-dimensional plots. We'll start with the contour plot, since that is the simplest. Each of the dimensional plots require you to pass them two square matrices that represent the x and y values. The functions then match up the x and y values element, and couple those to a third z matrix. If you are plotting a function, the easiest way to go about doing this is to use the `meshgrid` function. This function takes two separate 1 x n matrices (n must be the same for both matrices), replicates the first across n columns, the second across n rows, and outputs both resultant n x n matrices. It's much easier to show you than explain it, so here you go:

Example 2.5.3

```
%Ex_2_5_3

x = [1,2,3,4];
y = [5,6,7,8];

[X, Y] = meshgrid(x,y)
```

The output of this M-File looks like this:

```
X =
     1     2     3     4
     1     2     3     4
     1     2     3     4
     1     2     3     4
Y =
     5     5     5     5
     6     6     6     6
     7     7     7     7
     8     8     8     8
```

As you can see, this allows you to get every possible combination of x and y, which is your goal. These new matrices may then be acted upon to calculate a z matrix that you can plot with a 3d plotting function.

It's not generally useful to plot functions by manually creating matrices that include the points that you wish to plot. Thus, we will generally be using the syntax introduced in Section 2.3 that generates monotonically increasing matrices to plot 3d functions. Here is an example that plots $f(x,y) = x^2 + y^2$:

Example 2.5.4

```
%Ex_2_5_4

x = -4:.1:4;
y = -4:.1:4;

[X, Y] = meshgrid(x,y)

Z = X.^2 + Y.^2;

contour(X,Y,Z)
```

You'll see a nice contour plot pop up in the figure window; that's all there is to it.

You do the same exact thing with all of the other 3-dimensional plotting functions. Everything is the same, as that last example, except you change `contour` to whatever 3-dimensional graphing function that you would like. Like the `plot` function, there are many other ways to format the graph by passing arguments to the function. If you want to know how, either refer to the documentation or contact me.

There are many other visualization functions available to you through MATLAB, including histograms, vector fields (which we will be dealing with in the experiment itself), 3d vector fields, wire frame, 3d discrete points, etc. We will not cover these here, since they are not applicable to the lab that we are about to get to; refer to the documentation if you wish to learn about them.

Section 2.6: Importing Data

Importing data into MATLAB is very simple, so this will be a short section. There are two main ways to do it; you can use the import dialog, or you can use the `dlmread` function. Both require you to have your data in some standard ASCII spreadsheet format, such as tab separated or comma separated (as we discussed a little bit in the LabVIEW section).

The import dialog may be called from the Workspace window. It is a button at the top that looks like a folder with an arrow pointing out of it. If you click this, you will be presented with a wizard-type dialog that walks you through all the necessary steps of importing data.

The import dialog is useful if you just wish to do your work using the command window, but if you wish to script whatever you are doing, it is not so useful. The best thing to use in scripting situations is the `dlmread` function. This function accepts a file path as its first argument and the delimiter as the second argument. It then outputs a single matrix that represents whatever data it read. Make sure that whatever you're reading doesn't have anything but numbers in it; if it runs into something else, it will give an error.

To test the `dlmread` function, open up notepad (either find it buried wherever in the start menu, or go to Start > Run and type "notepad"), and create some sort of tab-separated data. Tab-separated data simply means that you designate each new column by a tab and each new row by a return. Alternatively, if you don't want to create it directly, you could open up Excel, type some data in, and tell it to save as tab-separated values.

Once you've created your data, do the following in the command window:

```
>> data = dlmread('C:\Your\file\path\yourfile.txt', '\t')
|| <MATLAB outputs whatever the contents of your file was>
```

It's that simple. You can script it also, of course (which is generally more useful). The only other important thing to remember is that you must find some way to get your data into an ASCII standard file (text file in other words); MATLAB doesn't (without an add-on anyways) support proprietary formats such as XLS. You can get it to work with Excel if you really want to, but you have to know what you are doing, and it is generally not worth the effort. If you really want to know how, search for "Excel" in the documentation.

Section 2.7: Closing Comments

As you have hopefully seen from what we have done so far, MATLAB is a very useful tool for data analysis. Once again, I have only scratched the surface of its ability; to really understand it, you need to just play around with it. Hopefully you are now familiar enough with it to understand the documentation, which is really the most important thing for our purposes, as far as the lab is concerned.

I didn't bother putting these examples up on the web site, since they were all so simple; however, if you would like them to be up, I do have them, and it wouldn't take very long to put them up. You will have considerably more MATLAB content, that is a little more complicated, available to you for use in the actual lab exercise that begins on the next page.

Feel free to contact me with any questions or comments that you may have; I will be more than happy to assist you any way that I can.

Lab

Section 3.1: Introduction

This lab's goal is to show you how to get computers to do the majority of your work for you. It will be setup a little different than normal, in that you are given free reign to do whatever you would like to do to reach your goal. What goal? Good question.

Goal: Use conducting paper, a SummaSketchIII, a power supply, a computer with LabVIEW and MATLAB, a NI USB-6008 DAQ, and physics to describe the electric field and potential present on the conducting when the power supply is hooked up to the two electrodes on the conducting paper.

The physics behind this experiment is purposefully very simple; the focus is on making your own experiment to show the results that you already know the answer (in great detail) to. In the following pages, I will be presenting you with information on the various aspects of the equipment you are using, as well as things in LabVIEW and MATLAB that I think will come in particularly handy. It is then up to you to put those things together to come up with something useful.

Because this is so open ended, and hence easy to get lost in details, I will be happy to provide any sort of help that I can. In fact, I will try to be in class fairly often, and if you ever request me to be there sometime during class, I will do my best to be there.

Section 3.2: Equipment

There are two key components to this experiment: the SummaSketchIII and the NI USB-6008. I will begin by explaining what the SummaSketchIII does, and how we will be using it.

The SummaSketchIII is a digitizer pad that has a modified stylus that allows it to measure voltages with its metal tip. We will be using the pad to allow us to simultaneously measure coordinates and couple those coordinates to a voltage. To do this, we will be placing the conducting paper on top of the SummaSketchIII and taking our measurements that way.

To use the SummaSketchIII, you simply touch the stylus to where you want to measure, and press down on the pen itself or on the little blue button on the top of it. This causes the SummaSketchIII to report data. This data is a short stream of bytes that is relatively complicated to explain. Luckily, you won't have to bother with that aspect of things; I have already written a driver that will handle it. All you have to do is drop my little SubVI on the block diagram, and voila--out pops coordinates whenever you need them. It is then your job to synchronize these coordinates with the voltage that is read though the DAQ.

To use the driver that I wrote, refer to the web site. There, you can find the documentation as well as the SubVI itself. It is under the "Lab Resources" area of the "Introduction to Modern Data Acquisition Lab" page.

The data acquisition device that we will be using is the NI USB-6008. It is basically a fancy, computer-controlled volt-meter. Thankfully, as we saw in the LabVIEW section on data acquisition, NI makes it really easy to control their instruments through LabVIEW using the DAQ-Assistant SubVI (see Section 1.8).

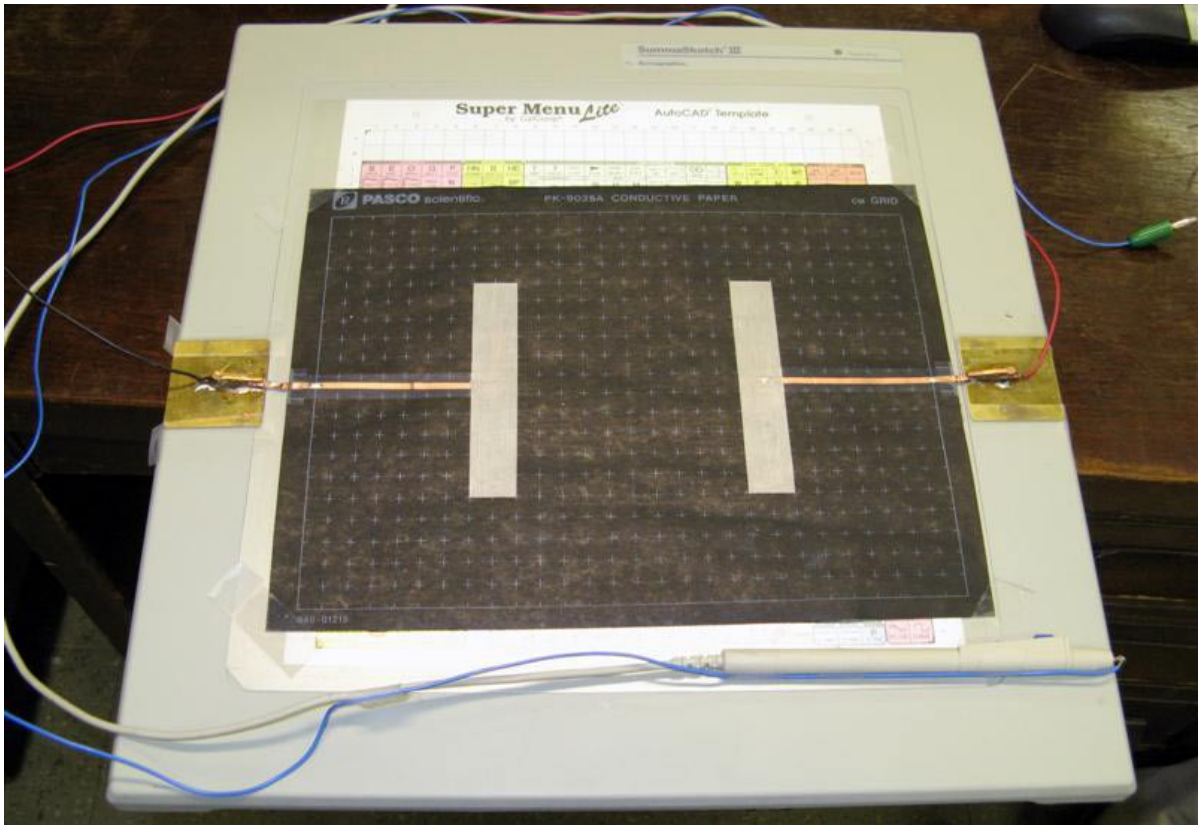
The USB-6008 will be used to measure the voltage that is present at the tip of the SummaSketchIII stylus. This will allows us, through LabVIEW, to take the different coordinates associated with physical locations on the conducting paper and assign a measured voltage to that position. With that data, you can proceed to do lots of interesting things.

Section 3.3: Goals

You already know the overall goal: to measure the voltage and electric field at multiple points on a piece of conducting paper. This goal may be further broken in to multiple "sub-goals"; I will proceed to do that now.

First off, you must plug in everything properly. Here is a picture of what the SummaSketchIII board should look like with the conducting paper, alligator clips, etc. hooked up.

Figure 3.3.1



Basically, all you must do is affix the conducting paper and alligator clips to the SummaSketchIII. You then connect the alligator clips to the power supply (set it to somewhere between +5V to +10V), and if you desire, patch the negative terminal to the ground terminal of the power supply. Since we will be using differential measurements with the DAQ, the ground patch shouldn't really matter, but you may do it if you would like.

Finally, you will want to hook the lead that comes out of the stylus to the positive terminal of the USB-6008, and the negative terminal of the USB-6008 should be patched to the negative terminal of the power supply (or ground if you patched the ground to the negative terminal). To make things simple, I would suggest using channel ai0 of the DAQ, but you may use whatever you wish to use.

Now you will want to test your setup. There are several ways you can test it, but the simplest

would probably be to turn the power supply on, navigate to MAX, run the test panels for the DAQ, and touch a few places on the conducting paper to get logical results. You may also want to directly start my "SummaSketchIII Get Coordinates" VI and see if the computer is getting coordinates properly. Do whatever testing you think is proper.

After all the equipment is properly setup, its time to begin constructing the program. Here is a breakdown of the core steps that you must take to take the measurement that we are looking for:

- Create a monitoring loop that waits until there are coordinates to be read from the SummaSketchIII (using the "Anything to Read?" output of my driver)
- Make this monitoring loop terminate when it finds that there are coordinates to read
- When the monitoring loop terminates, immediately read a voltage from the DAQ
- Find some way to store the results while you're taking the data (either pop it into an array or append it to a file)
- Put all of the above in a loop so that you can take multiple points
- Export the data to a file for storage
- Possibly use a MATLAB node to feed the data directly into MATLAB (that's up to you to decide)
- Use the data that you take to find out the electric field using whatever tools you wish (MATLAB is suggested)
- Do whatever you wish as far as displaying the results (plots, tables, histograms, whatever)

That's it! Now, in the next section, I will discuss a specifics about LabVIEW and MATLAB that may be useful to you.

Section 3.4: MATLAB and LabVIEW Tips

First, I will begin by describing how to go about doing one of the integral parts of designing this experiment: reading the coordinates. I will begin by describing a little bit about how the tablet works.

There are two main modes for the SummaSketchIII (from now on, I will refer to it as the SSIII): Stream Mode and Point Mode. Stream mode constantly reports data whenever the stylus is close enough to the pad to produce any data. It reports data at 30 reports per second by default, but you can change that behavior fairly easily. My driver will handle this mode, but it's not very useful given what we wish to use the SSIII for; we need point mode for our application. Point mode only takes data once for every push of the button or depression of the stylus. You can set the mode to point mode simply by running the VI entitled "SSIII - Set Point Mode." See the documentation on how to use that VI (it's very simple).

After you set the point mode, you may wish to configure the resolution of the pad. Here is a list of the possible resolutions:

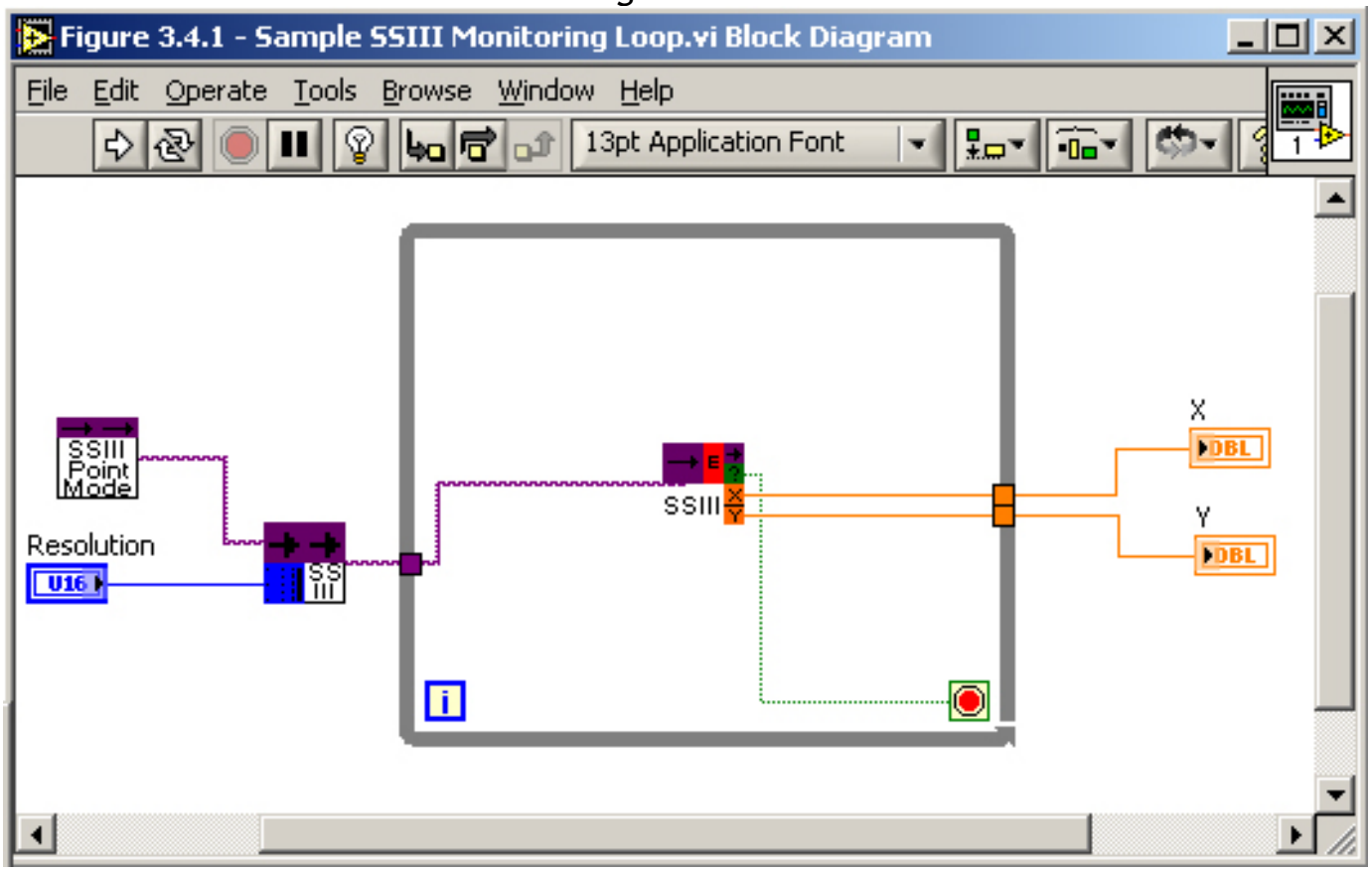
- 1 lpi
- 2 lpi
- 4 lpi
- 100 lpi
- 200 lpi
- 10 lpmm
- 400 lpi
- 500 lpi
- 20 lpmm
- 1000 lpi
- 40 lpmm

The resolution doesn't matter too much as far as accuracy is concerned; however, it is essential to know what the numbers that are being reported as coordinates mean. You must scale the dimensions according to what resolution that you select. This is essential if you wish to report the field in standard units of any kind.

After you have set the tablet to point mode and set the resolution, you are ready to take coordinate data. You use my VI called "SSIII - Get Coordinates" for that task. Once again, read the documentation to see how to go about using that VI.

Here is a way that you might go about taking data with the SSIII.

Figure 3.4.1



This VI begins by setting the SSIII to point mode. It then sets the resolution to whatever resolution is specified on the front panel. Finally, it enters a loop that terminates when the SSIII Driver reports that there are coordinates to read, and it outputs those coordinates when the loop terminates.

It could be made better by making use of the error reporting feature of the driver. For example, if an error is reported, you could make it so that the loop doesn't terminate and the serial buffer gets cleared. This lets you start over with a clean slate. Also, you could add a stop button, or any other set of features that you desire; hopefully you get the picture.

Now, instead of giving a detailed tutorial on how to do the contour plot with the vector plot superimposed upon it, I will present it in generalized steps and leave the particulars up to you and the MATLAB help function.

First, I will present you with the problem. As you saw in section 2.5, you have to have a grid of X and Y coordinates and you must have a Z value paired with every possible combination of them. In other words, if you have X = [1 2 3 4] and Y = [5 6 7 8], you need a z data point for (1,5), (1,6), (1,7), (1, 8), (2,5), (2, 6)... etc. But what do you do if you only have discrete points? Lets say you take an X-Y coordinate pair off of the SSIII, and then you assign a voltage to that coordinate. When you do that multiple times, you'll get lots of points that are not arranged in a grid pattern. For example, you might have one X-Y coordinate that is (3,6) and then another that is (5,19). In order to have it in grid form for plotting in MATLAB, you would need to have the points (3,19) and (5,6) as well. But you won't have those points. So our task now is to find a fancy way to plot it anyways.

Here are the steps:

- Take your data in the form of 3 vectors, X (x-coordinate), Y, (y-coordinate) and V (voltage)

- Use MATLAB's `linspace` function to generate a monotonically increasing set X Y data that starts at the minimum of your data's X and Y coordinates (use `min` and `max` to find the max and min of the coordinate data)
- Plug this new data into an interpolating function called `griddata` that will generate V values for the intermediate values; this will result in you having interpolated voltages associated with the linspace coordinates
- `meshgrid` the results
- Use the `gradient` function to take the gradient of the Voltage to get the electric field
- Evaluate the gradient at the `linspace` generated points
- Use `contour` to plot the original data
- Turn `hold` on so that you can put two plots on top of one another
- Use `quiver` to generate the vector (quiver) plot
- You should then have a contour plot that has the electric field vectors on it
- Turn `hold` off
- Do a `new figure` so you can pop up a new window
- Pass `surf` the `meshgrid`'ed `linspace` coordinates and the `griddata` voltages associated with them
- Voila, you're done

If you come up with any other big questions, feel free to ask me, and I will probably add them to this section.

Section 3.5: Closing Comments

Now it looks like you're on your own! Feel free to be creative with anything that you can think of to do with this. The things I've mentioned here are just the things that I thought of that you might want to do with something like this; there are many other possibilities. In fact, this was born out of a desire to rework the undergraduate lab on this subject, so if you can come up with any good, new ideas, by all means, go for it. A version of your idea may find its way into the undergraduate curriculum.

If you wish to ask me questions, I'll do my best to be in the lab on most days that you are. If not, I would be happy to arrange a time to meet with you outside of class; just let me know when would be a good time.

8.1.1 Critism

LabVIEW

It's proprietary and costs thousands of Euros per seat. The code and settings are hidden from plain site in all sorts menus and behind other widgets. Sometimes the wiring for Labview code can get too complex and becomes impossible to comment, debug, and make extensible for any future changes. A it is a nonlinear graphical programming (dataflow driven) interface it is hard do follow pograms. You have to use the mouse even at run time not only at development time. It has to rebuild all its data acquisition sub-VIs every time you want to make a tiny change to the sampling mode. Compiled executables produced by the Application Builder are not truly standalone in that they also require that the LabVIEW run-time engine be installed on any target computer on which users run the application. The use of standard controls requires a runtime library for any language and all major operating system suppliers supply the required libraries for common languages such as C. However, the runtime required for LabVIEW is not supplied with any operating system and is required to be specifically installed by the administrator. This requirement can cause problems if an application is distributed to a user who may be prepared to run the application but does not have the inclination or permission to install additional files on the host system prior to running the executable. NI hardware runs mostly on PC-based system-buses (PXI, PCI, ...) and depends on OS-DLLs.

MATLAB

It's proprietary and costs thousands of Euros per seat. Scientists mainly use MATLAB for their simulations, especially due to plenty of additional libraries and the Simulink addon. Libraries contain specific higherlevel functions of a particular field. Such functions speed up the development of advanced applications. As soon as a functional and intuitive graphical user interface or interaction with hardware (signal acquisition and generation) is required instead of MATLAB, LabVIEW is mostly used. Some more experienced users also choose a combination of both tools, which communicate via either the API or DLL libraries. Data can be exported from one and imported into another program for further processing. There is also a possibility to implement MATLAB code in LabVIEW. Most of MATLAB functions can be integrated into LabVIEW by using MathScriptfunction module. [22]

8.2 ROOT

Published on *ROOT* (http://root.cern.ch/drupal)

Architectural Overview

architecture [1] overview [2]

The backbone of the ROOT architecture is a layered class hierarchy with, currently, around 1200 classes [3] grouped in about 60 frameworks (libraries) divided in 19 main categories [4] (modules). This hierarchy is organized in a mostly single-rooted class library, that is, most of the classes inherit from a common base class **`TObject`** [5]. While this organization is not universally popular in C++, it has proven to be well suited for our needs (and indeed for almost all successful class libraries: Java, Smalltalk, MFC, etc.). It enables the implementation of some essential infrastructure inherited by all descendants of `TObject`. However, we also can have classes not inheriting from `TObject` when appropriate (e.g., classes that are used as built-in types, like `TString` [6]).

The Class Categories

The classes in the ROOT base category [7] provide the most low-level building blocks of ROOT. For example, the `TObject` class, which implements common behaviour for all ROOT classes. The class `TClass` [8] and its helper classes that provide support for extended runtime type information. The storage manager `TStorage` [9] which handles all memory allocation and de-allocation operations and performs basic error checking (memory overwrites, etc.). The class `TFile` [10] which provides a hierachical sequential and direct access persistant object store. The operating system abstraction layer `TSystem` [11] and the concrete OS interfaces `TUnixSystem` [12] and `TWinNTSystem` [13] concentrate all OS dependent behavior, like file system access, dynamic loading, error handling and networking for the different platforms supported by ROOT.

The classes in the container catagory [14] provide general purpose data structures like, arrays [15], lists [16], sets [17], B-trees [18], maps [19], etc., which are heavily used in the implementation of ROOT itself.

The physics category [20]provides classes that implement the Feldman Cousins [21] algorithm, N-body phase space generator [22], Lorentz rotation [23], Lorentz vector [24], 2-D vectors [25], 3-D vectors [26], etc.

The matrix category [27]provides classes for lazy matrices [28], general matrices [29] and vectors [30].

The histogram category [31]provides classes for advanced statistical data analysis, like 1D [32], 2D [33] and 3D [34] histograming of short, long, float or double values, with fixed or variable bin sizes, profile histograms and formula [35] evaluation.

The minimization catergory [36] provides classes that provide interfaces to Minuit [37], Fumili [38] and linear fitter [39] algorithms. [40]

The tree and ntuple category [41] contains the tree system [42]. The row-wise and column-wise ntuples have been one of the major strengths of the PAW system. Trees extend the concept of ntuples to all complex objects and data structures found in raw data, ESD's and AOD.s. The idea is that the same data model, same language, same style of queries can be used on all data sets in an experiment. Trees are designed to support not only complex objects, but also a very large number of them in a large number of files. Ntuples are simple trees with only simple types (int, float, double, etc.).

The 2D graphics category [43] provides classes for low-level graphics primitives, like lines [44], arrows [45], rectangles [46], ellipses [47], text [48], legends [49], annotations [50], text with Latex notation [51], splines [52], etc., but also the higher level constructs like pads [53] and canvases [54]. They also handle basic style [55] and attribute management.

The 3D graphics category [56]provides basic 3D graphics primitives, like 3D polylines [57] and 3D polymarkers [58] as well as higher level geometrical shapes (boxes [59], cones [60], polygons [61], tubes [62], etc.) which can be efficiently assembled into complex geometries [63].

The image processing category [64] provides classes such as TASImage [65] for image processing.

The detector geometry category [66] provides classes for building, browsing, tracking and visualizing detector geometries.

The code is independent from the Monte Carlo simulation packages and therefore it does not contain any constraints related to physics. However, the package defines a number of hooks for tracking, such as materials, magnetic field or track state flags, in order to allow interfacing to tracking MC's. The final goal is to be able to use the same geometry for several purposes, such as tracking, reconstruction or visualization, taking advantage of the ROOT features related to bookkeeping, I/O, histograming, browsing and GUI's. The geometrical modeler [67] is the most important component of the package and it provides answers to the basic questions like "Where am I ?" or "How far am I from the next boundary ?", but also to more complex ones like "How far am I from the closest surface ?" or "Which is the next crossing along a helix ?".

The neural network category [68] provides classes for multi layer perceptrons [69], etc.

The graphical user interface category [70] provides all the classes needed to build a modern, cross-platform, GUI. There are classes for many basic widgets such as buttons [71], windows [72], dialogs [73] and menus [74] and higher level widgets. *[75]*

The C++ interpreter [76], CINT, allows the construction of applications in which the user has to learn only one language, C++, to communicate with the system. The command language, macro language and programming language are all one and the same.

The networking category [77] provides classes for client [78] and server sockets [79] that allow to easily construct client/server applications.

The SQL interface [80] provides a simple, but powerful abstract interface to different SQL database servers (MySQL, SAPDB, PostgreSQL, Oracle).

The documentation classes [81] allow the creation of hyperized C++ header and source files, inheritance trees, class indices, macro's and session transcripts. Thanks to this facility almost everything in the ROOT system can be automatically documented and cross-referenced.

The TObject Class

Most ROOT classes are derived from the `TObject` class. The `TObject` class defines protocols (abstract methods) for comparing objects, for object inspection, for object I/O, for graphics hit detection and for object notification, to name just the most important ones.

The ROOT object I/O facility [82] supports the streaming of arbitrarily complex polymorphic data structures from memory to a buffer. This buffer can than be to stored in a ROOT binary machine-independent file [10], an XML file [83] or send over the network. This functionality is based on the abstract `Streamer` method, which is overridden in subclasses to stream an object's instance variables. Circular structures are linearized, and multiple references to the same object are restored properly. Storing pointers is implemented by an object table, which assigns a unique identifier to each transmitted object. This identifier can be transferred to other address spaces or to permanent storage.

The Class Dictionary and Object Run-time Support

Even with the upcoming run-time type identification (RTTI) extension for C++, the run-time system does not provide any information about the class structure, the instance variables or the member functions of an object. Consequently, an additional mechanism had to be introduced to gather this information, in order to support `InheritsFrom`, `Inspect` and `Dump` methods, the object I/O facility and the automatic documentation system. ROOT uses the approach of associating with each class (via a static pointer) a special object describing its structure. These descriptors are instances of the class `TClass` [8] which is itself a subclass of `TObject`. The `TClass` objects store the following information about a class:

- The name and title of a class.
- The size of an instance in bytes.
- Its parent class(es).
- The names, types and descriptions of its instance variables.
- The names and signatures of its member functions.
- A source code reference to the definition and implementation part of the class.
- The address of the class' object factory method used to create a new object.

Because the C++ run-time system gives no access to type and structure information, the ROOT system uses a dictionary generator called **CINT** [76]. **CINT** parses the class header files and generates a dictionary (in the form of a C++ function). To link the CINT generated dictionary function to a class the programmer only has to add two pre-processor macros to the code. One macro, `ClassDef` [84], must be placed in the class definition file and the other macro, `ClassImp` [84], in the implementation file.

Source URL: http://root.cern.ch/drupal/content/architectural-overview

Links:
[1] http://root.cern.ch/drupal/category/package-context/architecture
[2] http://root.cern.ch/drupal/category/package-context/overview
[3] http://root.cern.ch/root/html/ClassIndex.html
[4] http://root.cern.ch/root/html/index.html
[5] http://root.cern.ch/root/html/TObject.html
[6] http://root.cern.ch/root/html/TString.html
[7] http://root.cern.ch/root/html/CORE_BASE_Index.html
[8] http://root.cern.ch/root/html/TClass.html
[9] http://root.cern.ch/root/html/TStorage.html
[10] http://root.cern.ch/root/html/TFile.html
[11] http://root.cern.ch/root/html/TSystem.html
[12] http://root.cern.ch/root/html/TUnixSystem.html
[13] http://root.cern.ch/root/html/TWinNTSystem.h
[14] http://root.cern.ch/root/html/CORE_CONT_Index.html
[15] http://root.cern.ch/root/html/TObjArray.html
[16] http://root.cern.ch/root/html/TList.html
[17] http://root.cern.ch/root/html/THashTable.html
[18] http://root.cern.ch/root/html/TBtree.html
[19] http://root.cern.ch/root/html/TMap.html
[20] http://root.cern.ch/root/html/MATH_PHYSICS_Index.html
[21] http://root.cern.ch/root/html/TFeldmanCousins.html
[22] http://root.cern.ch/root/html/TGenPhaseSpace.html
[23] http://root.cern.ch/root/html/TLorentzRotation.html
[24] http://root.cern.ch/root/html/TLorentzVector.html
[25] http://root.cern.ch/root/html/TVector2.html
[26] http://root.cern.ch/root/html/TVector3.html
[27] http://root.cern.ch/root/html/MATH_MATRIX_Index.html
[28] http://root.cern.ch/root/html/TMatrixDLazy.html
[29] http://root.cern.ch/root/html/TMatrixD.html
[30] http://root.cern.ch/root/html/TVectorD.html
[31] http://root.cern.ch/root/html/HIST_HIST_Index.html
[32] http://root.cern.ch/root/html/TH1.html
[33] http://root.cern.ch/root/html/TH2.html
[34] http://root.cern.ch/root/html/TH3.html
[35] http://root.cern.ch/root/html/TF1.html
[36] http://root.cern.ch/root/html/MATH_MINUIT_Index.html
[37] http://root.cern.ch/root/html/TMinuit.html
[38] http://root.cern.ch/root/html/TFumili.html
[39] http://root.cern.ch/root/html/TLinearFitter.html
[40] http://root.cern.ch/drupal/html/QUADP_Index.html
[41] http://root.cern.ch/root/html/TREE_TREE_Index.html
[42] http://root.cern.ch/root/html/TTree.html#TTree:description
[43] http://root.cern.ch/root/html/GRAF2D_Index.html
[44] http://root.cern.ch/root/html/TLine.html
[45] http://root.cern.ch/root/html/TArrow.html
[46] http://root.cern.ch/root/html/TBox.html
[47] http://root.cern.ch/root/html/TEllipse.html
[48] http://root.cern.ch/root/html/TText.html
[49] http://root.cern.ch/root/html/TLegend.html
[50] http://root.cern.ch/root/html/TPaveLabel.html
[51] http://root.cern.ch/root/html/TLatex.html
[52] http://root.cern.ch/root/html/TSpline.html
[53] http://root.cern.ch/root/html/TPad.html
[54] http://root.cern.ch/root/html/TCanvas.html
[55] http://root.cern.ch/root/html/TStyle.html
[56] http://root.cern.ch/root/html/GRAF3D_Index.html
[57] http://root.cern.ch/root/html/TPolyLine3D.html
[58] http://root.cern.ch/root/html/TPolyMarker3D.html
[59] http://root.cern.ch/root/html/TBRIK.html
[60] http://root.cern.ch/root/html/TCONE.html
[61] http://root.cern.ch/root/html/TPGON.html
[62] http://root.cern.ch/root/html/TTUBE.html
[63] http://root.cern.ch/root/html/TGeometry.html
[64] http://root.cern.ch/root/html/GRAF2D_ASIMAGE_Index.html
[65] http://root.cern.ch/root/html/TASImage.html
[66] http://root.cern.ch/root/html/GEOM_Index.html
[67] http://root.cern.ch/root/html/TGeoManager.html
[68] http://root.cern.ch/root/html/MATH_MLP_Index.html
[69] http://root/html/TMLPAnalyzer.html
[70] http://root.cern.ch/root/html/GUI_Index.html
[71] http://root.cern.ch/root/html/TGButton.html
[72] http://root.cern.ch/root/html/TGWindow.html
[73] http://root.cern.ch/root/html/TGTransientFrame.html
[74] http://root.cern.ch/root/html/TGMenuBar.html
[75] http://root.cern.ch/drupal/html/GED_Index.html
[76] http://root.cern.ch/root/html/CINT_Index.html
[77] http://root.cern.ch/root/html/NET_Index.html
[78] http://root.cern.ch/root/html/TSocket.html

[79] http://root.cern.ch/root/html/TServerSocket.html
[80] http://root.cern.ch/root/html/TSQLServer.html
[81] http://root.cern.ch/root/html/HTML_Index.html
[82] http://root.cern.ch/drupal/inputoutput
[83] http://root.cern.ch/root/html/TXMLFile.html
[84] http://root.cern.ch/drupal/how-write-objects-file

8.3 TANGO

The TANGO control system is a free open source device-oriented controls toolkit for controlling any kind of hardware or software and building SCADA systems. It is used for controlling synchrotrons, lasers, physics experiments in over 20 sites. It is being actively developed by a consortium of research institutes.

TANGO is a distributed control system. It runs on a single machine as well as hundreds of machines. TANGO uses two network protocols the omniorb implementation of CORBA and Zeromq. The basic communication model is the client-server model. Communication between clients and servers can be synchronous, asynchronous or event driven. CORBA is used for synchronous and asynchronous communication and Zeromq is used for event-driven communication (since version 8 of TANGO).

TANGO is based on the concept of Devices. Devices implement object oriented and service oriented approaches to software architecture. The Device model in TANGO implements commands/methods, attributes/data fields and properties for configuring Devices. In TANGO all control objects are Devices. [23]

8.4 EPICS

EPICS is a set of software tools and applications which provide a software infrastructure for use in building distributed control systems to operate devices such as Particle Accelerators, Large Experiments and major Telescopes. Such distributed control systems typically comprise tens or even hundreds of computers, networked together to allow communication between them and to provide control and feedback of the various parts of the device from a central control room, or even remotely over the internet.

EPICS uses Client/Server and Publish/Subscribe techniques to communicate between the various computers. Most servers (called Input/Output Controllers or IOCs) perform real-world I/O and local control tasks, and publish this information to clients using the Channel Access (CA) network protocol. CA is specially designed for the kind of high bandwidth, soft real-time networking applications that EPICS is used for, and is one reason why it can be used to build a control system comprising hundreds of computers.

At the Advanced Photon Source, EPICS is used extensively within the control system for the accelerator itself as well as by many of the experimental beamlines. There are about 250 IOCs (mostly Motorola VME boards using MC680x0 and PowerPC CPUs that run vxWorks, but we have a growing number of IOCs now on Linux, MacOS and RTEMS) that directly or indirectly control almost every aspect of the machine operation, while 40 Sun workstations and servers in the control room provide higher level control and operator interfaces to the systems, and perform data logging, archiving and analysis.

A Channel Access Gateway allows engineers and physicists elsewhere in the building to examine the current state of the IOCs, but prevents them from making unauthorized adjustments to the running system. In many cases the engineers can make a secure internet connection from home to diagnose and fix faults without having to travel to the site.

EPICS has to be reliable, and provide facilities to ensure the resulting control system is maintainable and easily upgraded. Many of those 200 IOCs can cause the APS accelerator to dump the beam if they go wrong or stop working, and in some cases a wrong output could cause da-

mage to equipment which would cost many thousands of dollars and take days or even weeks to repair. Our IOCs have to run continuously for many months without being rebooted, thus the reliability of EPICS helps APS achieve its target of 95

Originally all EPICS IOCs had to run the vxWorks Real-Time Operating System from Wind River, but since 2004 it has been possible to run IOCs on GNU/Linux, Solaris, MS Windows, MacOS and RTEMS. Portable software is available that allows non-EPICS control systems to act as CA servers. CA clients have always been able to run on a wide range of computers and operating systems - most flavors of Unix, GNU/Linux, Windows, RTEMS and vxWorks.

EPICS is also the name of the collaboration of organizations that are involved in the software's development and use. It was originally written jointly by Los Alamos National Laboratory and Argonne National Laboratory, and is now used by many large scientific facilities throughout the world (see the EPICS Sites page for some of the major users). Development now occurs cooperatively between these various groups, with much sharing of I/O device support and client applications.

There is a mailing list called tech-talk where most EPICS-related discussions occur. The collaboration runs hands-on training courses to learn how to use EPICS, and once or twice a year holds conferences where the latest developments are presented and future work is discussed. [24]

8.4.1 EPICS Getting started

Chapter 2

Getting Started

2.1 Introduction

This chapter provides a brief introduction to creating EPICS IOC applications. It contains:

- Instructions for creating, building, and running an example IOC application.
- Instructions for creating, building, and executing example Channel Access clients.
- Briefly describes iocsh, which is a base supplied command shell.
- Describes rules for building IOC components.
- Describes makeBaseApp.pl, which is a perl script that generates files for building applications.
- Briefly discusses vxWorks boot parameters

This chapter will be hard to understand unless you have some familarity with IOC concepts such as record/device/driver support and have had some experience with creating ioc databases. Once you have this experience, this chapter provides most of the information needed to build applications. The example that follows assumes that EPICS base has already been built.

2.2 Example IOC Application

This section explains how to create an example IOC application in a directory `<top>`, naming the application `myexampleApp` and the ioc directory `iocmyexample`.

2.2.1 Check that `EPICS_HOST_ARCH` is defined

Execute the command:

```
echo $EPICS_HOST_ARCH          (Unix/Linux)
```

or

```
set EPICS_HOST_ARCH              (Windows)
```

This should display your workstation architecture, for example `linux-x86` or `win32-x86`. If you get an "Undefined variable" error, you should set `EPICS_HOST_ARCH` to your host operating system followed by a dash and then your host architecture, e.g. `solaris-sparc`. The perl script `EpicsHostArch.pl` in the base/startup directory has been provided to help set `EPICS_HOST_ARCH`.

2.2.2 Create the example application

The following commands create an example application.

```
mkdir <top>
cd <top>
<base>/bin/<arch>/makeBaseApp.pl -t example myexample
<base>/bin/<arch>/makeBaseApp.pl -i -t example myexample
```

Here, <arch> indicates the operating system architecture of your computer. For example, solaris-sparc. The last command will ask you to enter an architecture for the IOC. It provides a list of architectures for which base has been built.

The full path name to <base> (an already built copy of EPICS base) must be given. Check with your EPICS system administrator to see what the path to your <base> is. For example:

```
/home/phoebus/MRK/epics/base/bin/linux-x86/makeBaseApp.pl ...
```

Windows Users Note: Perl scripts must be invoked with the command perl <scriptname> on Windows. Perl script names are case sensitive. For example to create an application on Windows:

```
perl C:\epics\base\bin\win32-x86\makeBaseApp.pl -t example myexample
```

2.2.3 Inspect files

Spend some time looking at the files that appear under <top>. Do this *before* building. This allows you to see typical files which are needed to build an application without seeing the files generated by make.

2.2.4 Sequencer Example

The sequencer is now supported as an unbundled product. The example includes an example state notation program, sncExample.stt. As created by makeBaseApp the example is not built or executed.

Before sncExample.stt can be compiled, the sequencer module must have been built using the same version of base that the example uses.

To build sncExample edit the following files:

- configure/RELEASE – Set SNCSEQ to the location of the sequencer.
- iocBoot/iocmyexample/st.cmd – Remove the comment character # from this line:

  ```
  #seq sncExample, "user=<user>"
  ```

The Makefile contains commands for building the sncExample code both as a component of the example IOC application and as a standalone program called sncProgram, an executable that connects through Channel Access to a separate IOC database.

2.2.5 Build

In directory <top> execute the command

```
make
```

NOTE: On systems where GNU make is not the default another command is required, e.g. gnumake, gmake, etc. See you EPICS system administrator.

2.2.6 Inspect files

This time you will see the files generated by make as well as the original files.

2.2.7 Run the ioc example

The example can be run on vxWorks, RTEMS, or on a supported host.

- On a host, e.g. Linux or Solaris

```
cd <top>/iocBoot/iocmyexample
../../bin/linux-x86/myexample st.cmd
```

- vxWorks/RTERMS – Set your boot parameters as described at the end of this chapter and then boot the ioc.

After the ioc is started try some of the shell commands (e.g. `dbl` or `dbpr <recordname>`) described in the chapter "IOC Test Facilities". In particular run `dbl` to get a list of the records.

The iocsh command interpreter used on non-vxWorks IOCs provides a help facility. Just type:

```
help
```

or

```
help <cmd>
```

where `<cmd>` is one of the commands displayed by help. The help command accepts wildcards, so

```
help db*
```

will provide information on all commands beginning with the characters db. On vxWorks the help facility is available by first typing:

```
iocsh
```

2.3 Channel Access Host Example

An example host example can be generated by:

```
cd <mytop>
<base>/bin/<arch>/makeBaseApp.pl -t caClient caClient
make
```

(or gnumake, as required by your operating system)

Two channel access examples are provided:

`caExample`
: This example program expects a pvname argument, connects and reads the current value for the pv, displays the result and terminates. To run this example just type.

  `<mytop>/bin/<hostarch>/caExample <pvname>` where

  - `<mytop>` is the full path name to your application top directory.
  - `<hostarch>` is your host architecture.
  - `<pvname>` is one of the record names displayed by the `dbl` ioc shell command.

`caMonitor`
: This example program expects a filename argument which contains a list of pvnames, each appearing on a separate line. It connects to each pv and issues monitor requests. It displays messages for all channel access events, connection events, etc.

2.4 iocsh

Because the vxWorks shell is only available on vxWorks, EPICS base provides iocsh. In the main program it can be invoked as follows:

```
iocsh("filename")
```

or

```
iocsh(0)
```

If the argument is a filename, the commands in the file are executed and iocsh returns. If the argument is 0 then iocsh goes into interactive mode, i.e. it prompts for and executes commands until an exit command is issued.

This shell is described in more detail in Chapter 18, "IOC Shell" on page 249 FIXPAGEEREF.

On vxWorks iocsh is not automatically started. It can be started by just giving the following command to the vxWorks shell.

```
iocsh
```

To get back to the vxWorks shell just say

```
exit
```

2.5 Building IOC components

Detailed build rules are given in chapter "Epics Build Facility". This section describes methods for building most components needed for IOC applications. It uses excerpts from the `myexampleApp/src/Makefile` that is generated by makeBaseApp.

The following two types of applications can be built:

- Support applications

  These are applications meant for use by ioc applications. The rules described here install things into one of the following directories that are created just below `<top>`:

  `include`
  : C include files are installed here. Either header files supplied by the application or header files generated from `xxxRecord.dbd` or `xxxMenu.dbd` files.

  `dbd`
  : Each file contains some combination of `include`, `recordtype`, `device`, `driver`, and `registrar` database definition commands. The following are installed:

    - `xxxRecord.dbd` and `xxxMenu.dbd` files
    - An arbitrary `xxx.dbd` file
    - ioc applications install a file `yyy.dbd` generated from file `yyyInclude.dbd`.

  `db`
  : Files containing record instance definitions.

`lib/<arch>`
All source modules are compiled and placed in shared or static library (win32 dll)

- IOC applications

  These are applications loaded into actual IOCs.

2.5.1 Binding to IOC components

Because many IOC components are bound only during ioc initialization, some method of linking to the appropriate shared and/or static libraries must be provided. The method used for IOCs is to generate, from an `xxxInclude.dbd` file, a C++ program that contains references to the appropriate library modules. The following database definitions keywords are used for this purpose:

```
recordtype
device
driver
function
variable
registrar
```

The method also requires that IOC components contain an appropriate epicsExport statement. All components must contain the statement:

```
#include <epicsExport.h>
```

Any component that defines any exported functions must also contain:

```
#include <registryFunction.h>
```

Each record support module must contain a statement like:

```
epicsExportAddress(rset,xxxRSET);
```

Each device support module must contain a statement like:

```
epicsExportAddress(dset,devXxxSoft);
```

Each driver support module must contain a statement like:

```
epicsExportAddress(drvet,drvXxx);
```

Functions are registered using an `epicsRegisterFunction` macro in the C source file containing the function, along with a `function` statement in the application database description file. The makeBaseApp example thus contains the following statements to register a pair of functions for use with a subroutine record:

```
epicsRegisterFunction(mySubInit);
epicsRegisterFunction(mySubProcess);
```

The database definition keyword `variable` forces a reference to an integer or double variable, e.g. debugging variables. The `xxxInclude.dbd` file can contain definitions like:

```
variable(asCaDebug,int)
variable(myDefaultTimeout,double)
```

The code that defines the variables must include code like:

```
int asCaDebug = 0;
epicsExportAddress(int,asCaDebug);
```

The keyword `registrar` signifies that the epics component supplies a named registrar function that has the prototype:

```
typedef void (*REGISTRAR)(void);
```

This function normally registers things, as described in Chapter 21, “Registry” on page 317. The makeBaseApp example provides a sample iocsh command which is registered with the following registrar function:

```
static void helloRegister(void) {
    iocshRegister(&helloFuncDef, helloCallFunc);
}
epicsExportRegistrar(helloRegister);
```

2.5.2 Makefile rules

2.5.2.1 Building a support application.

```
# xxxRecord.h will be created from xxxRecord.dbd
DBDINC += xxxRecord
DBD += myexampleSupport.dbd

LIBRARY_IOC += myexampleSupport

myexampleSupport_SRCS += xxxRecord.c
myexampleSupport_SRCS += devXxxSoft.c
myexampleSupport_SRCS += dbSubExample.c

myexampleSupport_LIBS += $(EPICS_BASE_IOC_LIBS)
```

The `DBDINC` rule looks for a file `xxxRecord.dbd`. From this file a file `xxxRecord.h` is created and installed into `<top>/include`

The `DBD` rule finds `myexampleSupport.dbd` in the source directory and installs it into `<top>/dbd`

The `LIBRARY_IOC` statement states that a shared/static library should be created and installed into `<top>/lib/<arch>`.

The `myexampleSupport_SRCS` statements name all the source files that are compiled and put into the library.

The above statements are all that is needed for building many support applications.

2.5.2.2 Building the IOC application

The following statements build the IOC application:

```
PROD_IOC = myexample

DBD += myexample.dbd

# myexample.dbd will be made up from these files:
myexample_DBD += base.dbd
myexample_DBD += xxxSupport.dbd
myexample_DBD += dbSubExample.dbd

# <name>_registerRecordDeviceDriver.cpp will be created from <name>.dbd
myexample_SRCS += myexample_registerRecordDeviceDriver.cpp
myexample_SRCS_DEFAULT += myexampleMain.cpp
myexample_SRCS_vxWorks += -nil-

# Add locally compiled object code
```

```
myexample_SRCS += dbSubExample.c

#The following adds support from base/src/vxWorks
myexample_OBJS_vxWorks += $(EPICS_BASE_BIN)/vxComLibrary

myexample_LIBS += myexampleSupport
myexample_LIBS += $(EPICS_BASE_IOC_LIBS)
```

PROD_IOC sets the name of the ioc application, here called myexample.

The DBD definition myexample.dbd will cause build rules to create the database definition include file myexampleInclude.dbd from files in the myexample_DBD definition. For each filename in that definition, the created myexampleInclude.dbd will contain an include statement for that filename. In this case the created myexampleInclude.dbd file will contain the following lines.

```
include "base.dbd"
include "xxxSupport.dbd"
include "dbSubExample.dbd"
```

When the DBD build rules find the created file myexampleInclude.dbd, the rules then call dbExpand which reads myexampleInclude.dbd to generate file myexample.dbd, and install it into <top>/dbd.

An arbitrary number of myexample_SRCS statements can be given. Names of the form <name>_registerRecordDeviceDriver.cpp, are special; when they are seen the perl script registerRecordDeviceDriver.pl is executed and given <name>.dbd as input. This script generates the <name>_registerRecordDeviceDriver.cpp file automatically.

2.6 makeBaseApp.pl

makeBaseApp.pl is a perl script that creates application areas. It can create the following:

- <top>/Makefile
- <top>/configure – This directory contains the files needed by the EPICS build system.
- <top>/xxxApp – A set of directories and associated files for a major sub-module.
- <top>/iocBoot – A subdirectory and associated files.
- <top>/iocBoot/iocxxx – A subdirectory and files for a single ioc.

makeBaseApp.pl creates directories and then copies template files into the newly created directories while expanding macros in the template files. EPICS base provides two sets of template files: simple and example. These are meant for simple applications. Each site, however, can create its own set of template files which may provide additional functionality. This section describes the functionality of makeBaseApp itself, the next section provides details about the simple and example templates.

2.6.1 Usage

makeBaseApp has four possible forms of command line:

```
<base>/bin/<arch>/makeBaseApp.pl -h
```

Provides help.

```
<base>/bin/<arch>/makeBaseApp.pl -l [options]
```

List the application templates available. This invocation does not alter the current directory.

```
<base>/bin/<arch>/makeBaseApp.pl [-t type] [options] app ...
```

Create application directories.

```
<base>/bin/<arch>/makeBaseApp.pl -i -t type [options] ioc ...
```

Create ioc boot directories.

Options for all command forms:

`-b base`
: Provides the full path to EPICS base. If not specified, the value is taken from the EPICS_BASE entry in config/RELEASE. If the config directory does not exist, the path is taken from the command-line that was used to invoke makeBaseApp

`-T template`
: Set the template top directory (where the application templates are). If not specified, the template path is taken from the TEMPLATE_TOP entry in config/RELEASE. If the config directory does not exist the path is taken from the environment variable EPICS_MBA_TEMPLATE_TOP, or if this is not set the templates from EPICS base are used.

`-d`
: Verbose output (useful for debugging)

Arguments unique to `makeBaseApp.pl [-t type] [options] app ...`:

`app`
: One or more application names (the created directories will have "App" appended to this name)

`-t type`
: Set the template type (use the `-l` invocation to get a list of valid types). If this option is not used, type is taken from the environment variable EPICS_MBA_DEF_APP_TYPE, or if that is not set the values "default" and then "example" are tried.

Arguments unique to `makeBaseApp.pl -i [options] ioc ...`:

`ioc`
: One or more IOC names (the created directories will have "ioc" prepended to this name).

`-a arch`
: Set the IOC architecture (e.g. vxWorks-68040). If `-a arch` is not specified, you will be prompted.

2.6.2 Environment Variables:

`EPICS_MBA_DEF_APP_TYPE`
: Application type you want to use as default

`EPICS_MBA_TEMPLATE_TOP`
: Template top directory

2.6.3 Description

To create a new `<top>` issue the commands:

```
mkdir <top>
cd <top>
<base>/bin/<arch>/makeBaseApp.pl -t <type> <app> ...
<base>/bin/<arch>/makeBaseApp.pl -i -t <type> <ioc> ...
```

makeBaseApp does the following:

- EPICS_BASE is located by checking the following in order:
  - If the -b option is specified its value is used.
  - If a <top>/configure/RELEASE file exists and defines a value for EPICS_BASE it is used.
  - It is obtained from the invocation of makeBaseApp. For this to work, the full path name to the makeBaseApp.pl script in the EPICS base release you are using must be given.
- TEMPLATE_TOP is located in a similar fashion:
  - If the -T option is specified its value is used.
  - If a <top>/configure/RELEASE file exists and defines a value for TEMPLATE_TOP it is used.
  - If EPICS_MBA_TEMPLATE_TOP is defined its value is used.
  - It is set equal to <epics_base>/templates/makeBaseApp/top
- If -l is specified the list of application types is listed and makeBaseApp terminates.
- If -i is specified and -a is not then the user is prompted for the IOC architecture.
- The application type is determined by checking the following in order:
  - If -t is specified it is used.
  - If EPICS_MBA_DEF_APP_TYPE is defined its value is used.
  - If a template defaultApp exists, the application type is set equal to default.
  - If a template exampleApp exists, the application type is set equal to example.
- If the application type is not found in TEMPLATE_TOP, makeBaseApp issues an error and terminates.
- If Makefile does not exist, it is created.
- If directory configure does not exist, it is created and populated with all the configure files.
- If -i is specified:
  - If directory iocBoot does not exist, it is created and the files from the template boot directory are copied into it.
  - For each <ioc> specified on the command line a directory iocBoot/ioc<ioc> is created and populated with the files from the template (with ReplaceLine() tag replacement, see below).
- If -i is NOT specified:
  - For each <app> specified on the command line a directory <app>App is created and populated with the directory tree from the template (with ReplaceLine() tag replacement, see below).

2.6.4 Tag Replacement within a Template

When copying certain files from the template to the new application structure, makeBaseApp replaces some predefined tags in the name or text of the files concerned with values that are known at the time. An application template can extend this functionality as follows:

- Two perl subroutines are defined within makeBaseApp:

  ReplaceFilename
  This substitutes for the following in names of any file taken from the templates.

  ```
  _APPNAME_
  _APPTYPE_
  ```

`ReplaceLine`
This substitutes for the following in each line of each file taken from the templates:

```
_USER_
_EPICS_BASE_
_ARCH_
_APPNAME_
_APPTYPE_
_TEMPLATE_TOP_
_IOC_
```

- If the application type directory has a file named `Replace.pl`, it can:
  - Replace one or both of the above subroutines with its own versions.
  - Add a subroutine `ReplaceFilenameHook($file)` which is called at the end of `ReplaceFilename`.
  - Add a subroutine `ReplaceLineHook($line)` which is called at the end of `ReplaceLine`.
  - Include other code which is run after the command line options are interpreted.

2.6.5 makeBaseApp templetes provided with base

2.6.5.1 support

This creates files appropriate for building a support application.

2.6.5.2 ioc

Without the `-i` option, this creates files appropriate for building an ioc application. With the `-i` option it creates an ioc boot directory.

2.6.5.3 example

Without the `-i` option it creates files for running an example. Both a support and an ioc application are built. With the `-i` option it creates an ioc boot directory that can be used to run the example.

2.6.5.4 caClient

This builds two Channel Access clients.

2.6.5.5 caServer

This builds an example Portable Access Server.

2.7 vxWorks boot parameters

The vxWorks boot parameters are set via the console serial port on your IOC. Life is much easier if you can connect the console to a terminal window on your workstation. On Linux the 'screen' program lets you communicate through a local serial port; run `screen /dev/ttyS0` if the IOC is connected to `ttyS0`.

The vxWorks boot parameters look something like the following:

```
boot device             : xxx
processor number        : 0
host name               : xxx
file name               : <full path to board support>/vxWorks
inet on ethernet (e)    : xxx.xxx.xxx.xxx:<netmask>
host inet (h)           : xxx.xxx.xxx.xxx
user (u)                : xxx
ftp password (pw)       : xxx
flags (f)               : 0x0
target name (tn)        : <hostname for this inet address>
startup script (s)      : <top>/iocBoot/iocmyexample/st.cmd
```

The actual values for each field are site and IOC dependent. Two fields that you can change at will are the vxWorks boot image and the location of the startup script.

Note that the full path name for the correct board support boot image must be specified. If bootp is used the same information will need to be placed in the bootp host's configuration database instead.

When your boot parameters are set properly, just press the reset button on your IOC, or use the `@` command to commence booting. You will find it VERY convenient to have the console port of the IOC attached to a scrolling window on your workstation.

2.8 RTEMS boot procedure

RTEMS uses the vendor-supplied bootstrap mechanism so the method for booting an IOC depends upon the hardware in use.

2.8.1 Booting from a BOOTP/DHCP/TFTP server

Many boards can use BOOTP/DHCP to read their network configuration and then use TFTP to read the applicaion program. RTEMS can then use TFTP or NFS to read startup scripts and configuration files. If you are using TFTP to read the startup scripts and configuration files you must install the EPICS application files on your TFTP server as follows:

- Copy all `db/xxx` files to `<tftpbase>/epics/<target_hostname\>/db/xxx`.
- Copy all `dbd/xxx` files to `<tftpbase>/epics/<target_hostname>/dbd/xxx`.
- Copy the `st.cmd` script to `<tftpbase>/epics/<target_hostname>/st.cmd`.

Use DHCP site-specific option 129 to specify the path to the IOC startup script.

2.8.2 Motorola PPCBUG boot parameters

Motorola single-board computers which employ PPCBUG should have their 'NIOT' parameters set up like:

```
Controller LUN =00
Device LUN     =00
Node Control Memory Address =FFE10000
Client IP Address       ='Dotted-decimal' IP address of IOC
Server IP Address       ='Dotted-decimal' IP address of TFTP/NFS server
Subnet IP Address Mask ='Dotted-decimal' IP address of subnet mask (255.255.255.0 for class C subnet)
Broadcast IP Address    ='Dotted-decimal' IP address of subnet broadcast address
Gateway IP Address      ='Dotted-decimal' IP address of network gateway (0.0.0.0 if none)
```

```
Boot File Name          =Path to application bootable image (..../bin/RTEMS-mvme2100/test.boot)
Argument File Name      =Path to application startup script (..../iocBoot/ioctest/st.cmd)
Boot File Load Address          =001F0000 (actual value depends on BSP)
Boot File Execution Address     =001F0000 (actual value depends on BSP)
Boot File Execution Delay       =00000000
Boot File Length                =00000000
Boot File Byte Offset           =00000000
BOOTP/RARP Request Retry        =00
TFTP/ARP Request Retry          =00
Trace Character Buffer Address =00000000
```

2.8.3 Motorola MOTLOAD boot parameters

Motrola single-board computers which employ MOTLOAD should have their network 'Global Environment Variable' parameters set up like:

`mot-/dev/enet0-cipa`='Dotted-decimal' IP address of IOC
`mot-/dev/enet0-sipa`='Dotted-decimal' IP address of TFTP/NFS server
`mot-/dev/enet0-snma`='Dotted-decimal' IP address of subnet mask (255.255.255.0 for class C subnet)
`mot-/dev/enet0-gipa`='Dotted-decimal' IP address of network gateway (omit if none)
`mot-/dev/enet0-file`=Path to application bootable image (..../bin/RTEMS-mvme5500/test.boot)
`rtems-client-name`=IOC name (mot-/dev/enet0-cipa will be used if this parameter is missing)
`rtems-dns-server`='Dotted-decimal' IP address of domain name server (omit if none)
`rtems-dns-domainname`=Domain name (if this parameter is omitted the compiled-in value will be used)
`epics-script`=Path to application startup script (..../iocBoot/ioctest/st.cmd)

The `mot-script-boot` parameter should be set up like:

```
tftpGet -a4000000 -cxxx -sxxx -mxxx -gxxx -d/dev/enet0
        -f..../bin/RTEMS-mvme5500/test.boot
netShut
go -a4000000
```

where the `-c`, `-s`, `-m` and `-g` values should match the cipa, sipa, snma and gipa values, respectively and the `-f` value should match the file value.

2.8.4 RTEMS NFS access

For IOCs which use NFS for remote file access the EPICS initialization code uses the startup script pathname to determine the parameters for the initial NFS mount. If the startup script pathname begins with a '`/`' the first component of the pathname is used as both the server path and the local mount point. If the startup script pathname does not begin with a '`/`' the first component of the pathname is used as the local mount point and the server path is "`/tftpboot/`" followed by the first component of the pathname. This allows the NFS client used for EPICS file access and the TFTP client used for bootstrapping the application to have a similar view of the remote filesystem.

2.8.5 RTEMS 'Cexp'

The RTEMS 'Cexp' add-on package provides the ability to load object modules at application run-time. If your RTEMS build includes this package you can load RTEMS IOC applications in the same fashion as vxWorks IOC applications.

8.5 Control System Studio

Control System Studio is an Eclipse-based collection of tools to monitor and operate large scale control systems, such as the ones in the accelerator community. It's a product of the collaboration between different laboratories and universities. [25]

9 Anhang

9.1 Literaturverzeichnis

Literaturverzeichnis

[1] Heidepriem, Jürgen [Auth.].,Prozessinformatik, 1: Grundzüge der Informatik.,München,2000,Oldenbourg

[2] Heidepriem, Jürgen [Auth.].,Prozessinformatik, 2: Prozessrechentechnik und Automationssysteme.,2. Aufl.,München,2004,Oldenbourg

[3] Faerber, Georg [Auth.].,Prozessrechentechnik : Grundlagen, Hardware, Echtzeitverhalten, 2., völlig neubearb. und erw. Aufl.,Berlin [u.a.],1992,Springer

[4] Jacobson, Erik [Auth.].,Einführung in die Prozessdatenverarbeitung,2., grundl. überarb. und erw. Aufl.,München [u.a.],1996,Hanser

[5] Halang, Wolfgang A. [Auth.]. Konakovsky, Rudolf [Auth.].,Sicherheitsgerichtete Echtzeitsysteme,München [u.a.],1999,Oldenbourg

[6] Ghassemi-Tabrizi, Ataeddin [Auth.].,Realzeit-Programmierung : mit 261 Abbildungen,Berlin [u.a.],2000,Springer

[7] Jansen, Werner [Compos.].,Grundkurs Feldbustechnik : INTERBUS ; Automatisierungstechnik nach IEC 61 158,Würzburg,2000,Vogel

[8] www.karakas-online.de/teia/JAVA/java_1_1_2.htm

[9] Gabler Wirtschaftslexikon (1997), Gabler Verlag, Wiebaden

[10] Gärtner, K.-P., Optimaler Automatisierungsgrad von Mensch-Maschine-Systemen,1993, DGLR Bericht 94-01

[11] Grube, U., Zur Bestimmung des Automatisierungsgrades für die Planung von Fertigungssystemen, 1990,Jochem Heizmann Verlag, Karlsruhe

[12] Henning, K., Mensch und Automatisierung,1990, Westdeutscher Verlag, Opladen

[13] Hesse, S., Fertigungsautomatisierung, 2000,Vieweg Verlag Braunschweig/Wiesbaden

[14] Schraft, R.D., Automatisierung der Produktion, 1998,Springer Verlag, Berlin

[15] Timpe, K.-P., Mensch-Maschine-Systemtechnik, 2002,Symposion Publishing, Düsseldorf

[16] Bolch, G.,Universität Erlangen-Nürnberg, Skript zur Prozessautomatisierung

[17] U.Tietze/ Ch. Schenk, Halbleiter Schaltungstechnik, 1999, Springer Verlag Berlin, 11. Aufl.,Kapitel 21/ 22

[18] Udo Zölzer, Digitale Audiosignalverarbeitung, 1997, B.G. Teubner Verlag, 2. Aufl., Kapitel 2.1/3.2/3.3

[19] M. Arnold, Kommunikationskonzept für die Prozessleittechnik, Verlag Mainz, Wissenschaftsverlag Aachen 1999, Dissertation RWTH Aachen 1998.

[20] G. Schnell (edit.), Bussysteme in der Automatisierungstechnik, 1994,Vieweg

[21] Matt Hollingsworth, Intro modern Daq, http://mhworth.com/static/projects/intro-modern-daq/imda.pdf

[22] Tadey Tasner et. al., COMPARISON OF LabVIEW AND MATLAB FOR SCIENTIFIC RESEARCH, ANNALS OF FACULTY ENGINEERING HUNEDOARA - INTERNATIONAL JOURNAL of ENGINEERING Tome X (Year 2012) FASCICULE 3 (ISSN 1584 - 2673), 2012

[23] http://www.tango-controls.org

[24] http://www.aps.anl.gov/epics/about.php

[25] http://controlsystemstudio.org

[26] http://163.25.99.51/Huang_Computer/Programming Language-website2.pdf

[27] http://profesores.fi-b.unam.mx/cintia/l922.pdf