**5.3 Data Flow-Oriented Design**

Data flow- oriented design is an architectural design method that allows a transition from the analysis model to design description of program structure. It is useful when information is processed sequentially and no hierarchical data structure exists, for example, microprocessor control application and numerical analysis procedure. The transition from information flow (represented as a DFD) to program structure is accomplished as part of a six-step process:

1. The type of information flow is established.
2. Flow boundaries are indicated.
3. The DFD is mapped into program structure.
4. Control hierarchy is defined.
5. Resultant structure is refined using design measures and heuristics.
6. The architectural description is refined and elaborated.

The information flow type is the driver for the mapping approach (which is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a program structure) required in step 3. There are two flow types:-

**1. Transform Flow**

Information must enter and exit software in an external world. For example, data typed on a keyboard, tones on a telephone line, and pictures on a computer graphics display are all forms of external world information. Such externalized data must be converted into an internal form for processing. Consider the figure 5.2:

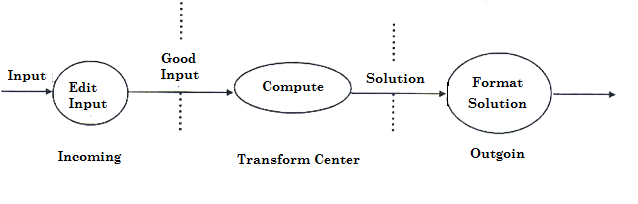


Figure 5.2 the Transform Flow

Information enters the system along paths that transform external data into an internal form and will be identified as ***incoming flow***. Incoming data are passed through a ***transform center*** and begin to move along paths that lead of the software. Data moving along these paths are called ***outgoing flow***. The overall flow of data occurs in a sequential manner. When a segment of a data flow diagram has these characteristics, transform flow is present.

**2. Transaction Flow**

Information flow is characterized by a single data item, called a **transaction** that triggers other data flow along one of many paths. When a DFD takes the forms shown in figure 5.3, transaction flow is present.

Transaction flow is characterized by data moving along an incoming path that converts external world information into a transaction. The transaction is evaluated and based on its value; flow along one of many action paths is initiated. The hub of information flow from which many action paths emanate is called a **transaction center**.

It should be noted that within **DFD** for a large system, both transform and transaction flow may be present. For example, in a transaction flow, information flow along an action path may have transform flow characteristics.

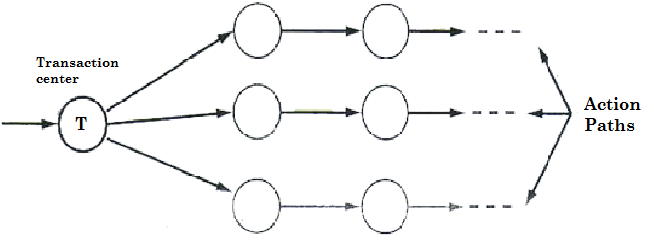
****

Figure 5.3 the Transaction Flow

**5.4 Design of Real-Time system**

The design of real-time software must incorporate all of the fundamental concepts associated with high- quality software. In addition, real-time software poses a set of unique problems for the designer:

1. Representation of interrupt and context switching.
2. Concurrency as manifested by multitasking and multiprocessing.
3. Inter task communication and synchronization.
4. Wide variations in data and communication rates.
5. Representation of timing constraints.
6. Special requirements for error handling and fault recovery.
7. Asynchronous processing.
8. Necessary and unavoidable coupling with operating system, hardware, and other external system elements.

Data flow-oriented design techniques provides a foundation for real-time design. To support real-time design, data flow methods must be extended by providing: (1) a mechanism for representing task communication and synchronization, (2) a notation for representing state dependency, and (3) an approach to connect conventional data flow methods to the real time world.

Because most real-time systems spawn multiple tasks that either share a single processor or execute simultaneously in distributed processor, mechanisms must be available to synchronize tasks and provide communication between tasks. **Synchronization** occurs through mutual exclusion or cross stimulation. **Mutual exclusion** is applied when two tasks may access a shared data area at the same time. **Semaphores** are used to exclude one task from accessing the data while another is using (reading or writing) the same data. **Cross stimulation** is implemented when one task signals another (waiting) task that it has completed some activity and the signaled task may proceed.

**Task communication** occurs when one task must transmit information to another task. Message communication is a common communication approach. When a producer task sends a message to a consumer task and then waits for a response from the consumer task, communication is **closely coupled**. When producer and consumer tasks continue processing at their own rates and use a message queue to buffer messages, communication is **loosely coupled**.