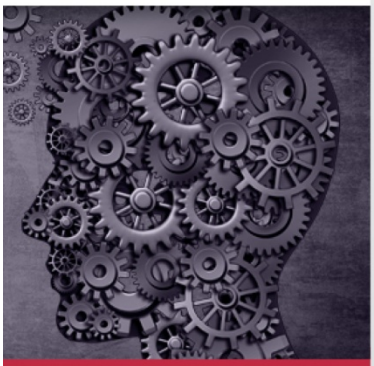

# Data Responsive Modular Interleaved Task Programming System

*An IP.com Prior Art Database Technical Disclosure*

Authors et. al.: IBM
Morrison, JP

Original Publication Date: January 01, 1971
Original Disclosure Information: TDB 01-71 p2425-2426
IP.com Number: IPCOM000073741D
IP.com Electronic Publication: February 23, 2005

**Data Responsive Modular Interleaved Task Programming System**

Reduced programming costs and earlier productivity of new programmers are achieved with a composite data processing program consisting of an assembly of prewritten and custom-written MODULES such as 3-8, communicating by data elements passing through QUEUES such as 11-15, which can each store some number (called its CAPACITY) of data elements. Each module (e.g. 6) is a program which performs a data processing task such as COLLATE, PRINT REPORTS, READ TAPE, and EDIT CARDS on a data element presented to it by SCHEDULER 20 via a queue (e.g. 13). Scheduler 20 activates modules (e.g. 6) on the basis of availability of data elements for processing by the modules, service requests by the modules for data handling, and checks for completion of external events.

Sketch A is a FLOW DIAGRAM representation of a typical program formed by assembling a selection of modules 3-8 into a customized array joined by a network of queues 11-15, which are operated on and activated by scheduler 20. 16-19 represent queues which are LOCAL to particular modules (e.g. 5,6). If the capacity of such a local queue is 1, it is called a SHUNT (e.g. 16).

To execute his program, the programmer codes one statement for each module in the flow diagram, the modules to be custom-written, and constant information in core storage representing selections among options designed into the prewritten modules. These are COMPILED, then LINK-EDITED with the prewritten modules and scheduler 20 to form an executable program. Modules 3, 4, 7, and 8 are connected to external media represented by 1, 2, 9, and 10, respectively, by the above constant information.

In sketch B, Q1 and Q3 are the queues by which modules M1, M2 and MP communicate. Q2 is local to M2. M1 and M2 are examples of prewritten modules and are made available to the programmer in a library 21, which also contains scheduler 20. MP is a typical module custom-written by the programmer. A typical sequence of operation of scheduler 20 might be as follows: scheduler 20 removes a data element from Q1, and presents it to M2 for processing. After processing, M2 requests scheduler 20 to transmit this data element to Q3. Scheduler 20 then determines whether Q3 is full, and, if not, places the data element in it and returns control to M2. If Q3 is full, however, M2 is suspended, and control given to another module (e.g. MP), provided its input queue, Q3, is not empty, or to some other suspended module provided it can now proceed. M2 might alternatively request scheduler 20 to place a data element in its local queue Q3, or to retrieve one from it. If scheduler 20 cannot do this, it will return an error indication to M2, and no suspension of M2 occurs.

Other typical functions supported by scheduler 20 are: get the next data element from a designated queue (e.g. Q1); allocate core storage for (create) a new data element; release core storage allocated to (destroy) a data element; suspend a module (e.g. M1) until completion of an external event such as completion of a write operation onto disc.

Data elements represent objects (e.g. automobiles, employees) handled by the program and fall into CLASSES determined by their characteristics, such as

amount of storage required for an element (typically 80 bytes). Classes are divided into DYNAMIC and STATIC. Dynamic data elements are "created", are passed from one module to another for processing and possible modification and are finally "destroyed". Static data elements are unmodifiable, and cannot be "created" or "destroyed". Since data elements do not move in core storage, it is the references to data elements which are moved through the queues by scheduler 20. This feature allows one to program as though multiple copies of a static data element are immediately available when needed.